

Классы



Владимир
Чебукин



Владимир Чебукин

Frontend-разработчик



[Владимр Чебукин](#)



План занятия

1. [Объектно-ориентированное программирование](#)
2. [class, создание классов, конструктор](#)
3. [Методы, get, set](#)
4. [Наследование и super](#)
5. [Статические методы](#)
6. [Нововведения последних версий ES](#)



Классы

ООП

ООП — методология разработки программ, в которой все важные **вещи** представляются **объектами**.

Каждый объект построен по определённым правилам, которые называют **классом**.

Классы основываются друг на друге, что называют **наследованием**



4 кита

Четыре принципа, поверх которых строятся объектно-ориентированные приложения:

- **абстракция** — рассмотрение объекта реального мира в контексте конкретной задачи
- **инкапсуляция** — сокрытие внутренней реализации
- **наследование** — передача характеристик одних объектов другим через отношение «является»: *кот является животным*
- **полиморфизм** — возможность работать с конкретной структурой данных как с абстрактной

Пример объектно-ориентированного решения: демо

ООП стиль — это не обязательно меньше строк кода в сравнении с другими парадигмами, но зачастую **лучшее понимание и чтение кода**:

```
class Cart {  
  
    constructor() {  
        // внутреннее хранилище  
        this.items = [];  
    }  
  
    find(product) {  
        return this.items.find((cartItem) => cartItem.product.id === product.id);  
    }  
}
```

Продолжение кода на следующей странице

Пример объектно-ориентированного решения: демо

```
add(product) {  
    const item = this.find(product);  
  
    if (item) {  
        item.quantity++;  
        return;  
    }  
  
    this.items.push({  
        product,  
        quantity: 1,  
    });  
}
```

Продолжение кода на следующей странице

Пример объектно-ориентированного решения: демо

```
class Book {  
  
    constructor(id, title, price) {  
        this.id = id;  
        this.title = title;  
        this.price = price;  
    }  
}  
  
const cart = new Cart();  
const bookOne = new Book(11, "Приключения Тома Сойера", 300);  
const bookTwo = new Book(12, "Краткая история времени", 400);  
  
cart.add(bookOne);  
cart.add(bookTwo);
```

Продолжение кода на следующей странице

Преимущества ООП стиля

- Идеально **подходит для большого количества** типовых объектов
- Позволяет **удобно делить сложные конструкции** на мелкие составляющие
- Упрощает работу с **внутренним состоянием**



class

В ES6 добавилась **новая конструкция** — *class*. Класс представляет собой **макет**, по которому будет создан конкретный объект. Точно также, как по чертежу самолёта делают самолёт.

```
class Aircraft {  
}
```

new

new создаёт по чертежу класса **экземпляр**:

```
class BMW {  
}  
  
const bmw1 = new BMW();  
const bmw2 = new BMW();
```

Мы создали **два экземпляра** типа BMW. Это два разных **объекта**. Как и в реальной жизни, два автомобиля одной серии в итоге всё равно получаются немного разными, со своей душой.

Конструкция **new** всегда возвращает объект

Классы и функции-конструкторы

В JavaScript **конструкция class** — удобное сокращение существовавших ранее подходов для создания объектов и инкапсуляции логики:

```
class BMW {  
  
}  
  
function BMW {  
  
}
```

Классы очень похожи на **функции-конструкторы**, но есть ряд отличий, о которых можно подробнее прочитать:

<https://learn.javascript.ru/class#ne-prosto-sintaksicheskiy-sahar>.

Экземпляры — обычные объекты

Экземпляры типов, заданных конструкцией **class**, — обычные объекты. Им также можно задавать свойства и методы:

```
const obj = {};  
obj.title = "Я - обычный объект!";  
obj.showTitle = function () {  
    console.log(this.title);  
};  
  
class SuperObject { }  
const superObj = new SuperObject();  
  
// аналогично  
superObj.title = "Супер";  
superObj.showTitle = function () {  
    console.log(this.title);  
};
```

Конструктор класса

Для **гибкой настройки объектов** им можно передавать начальные параметры.

Например, было бы здорово задать название книги для класса *Book*:

```
class Book {  
  
}  
  
const book = new Book('Понедельник начинается в субботу');
```

Но как воспользоваться этим значением?

Конструктор класса

Для этого существуют **конструкторы класса** — функции, которые будут запущены в момент создания экземпляра объекта:

```
class Book {  
  
    constructor(name) {  
        console.log(`Вы хотите создать книгу с названием «${name}»`);  
    }  
}  
  
const book = new Book('Понедельник начинается в субботу');
```

Было бы здорово сохранить получаемое значение в свойство экземпляра

this

Эту проблему решает старый знакомый *this*, доступный в **конструкторе**.

this всегда указывает на создаваемый экземпляр:

```
class Book {  
    constructor(name) {  
        this.name = name;  
    }  
}  
  
const book = new Book('Понедельник начинается в субботу');  
  
console.log(book.name); // 'Понедельник начинается в субботу'
```

this

Вот вариант для того, чтобы упорядочить простую телефонную книгу:

```
class Person {
  constructor(firstName, lastName, phone) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.phone = phone;
  }
}

const phonebook = [
  new Person('Владислав', 'Иванов', '+74993412233'),
  new Person('Леонида', 'Петрова', '+74993412232'),
];
```

Магия конструктора

Благодаря *this* и **конструктору** мы имеем возможность добавлять **шаблонные свойства**, делая код более лаконичным. Сравните:

1

```
1  const oleg = {
2    name: 'Олег',
3    lastName: 'Иванов',
4    gender: 'male',
5    type: 'human'
6  }
7
8  const ivan = {
9    name: 'Иван',
10   lastName: 'Широков',
11   gender: 'male',
12   type: 'human'
13 }
14
15 const nikita = {
16   name: 'Никита',
17   lastName: 'Огурцов',
18   gender: 'male',
19   type: 'human'
20 }
```

2

```
1  class Male {
2    constructor(name, lastName) {
3      this.name = name;
4      this.lastName = lastName;
5      this.gender = 'male';
6      this.type = 'human';
7    }
8  }
9
10 const oleg = new Male('Олег', 'Иванов');
11 const ivan = new Male('Иван', 'Широков');
12 const nikita = new Male('Никита', 'Огурцов');
```

Магия конструктора

В результате мы смогли без особого труда добавить **два дополнительных свойства**, которые будут присутствовать абсолютно во всех экземплярах объекта:

```
class Male {  
  
    constructor(name, lastName) {  
        this.name = name;  
        this.lastName = lastName;  
        this.gender = "male";  
        this.type = "human";  
    }  
}  
  
const oleg = new Male("Олег", "Иванов");  
const ivan = new Male("Иван", "Широков");  
const nikita = new Male("Никита", "Огурцов");
```

typeof. Никакой магии

Классы в JS — это конструкция, пришедшая с ES6, делает жизнь разработчика удобнее.

Давайте посмотрим, что из себя представляет *Object* и *Array* в конструкциях:

```
// тут мы не пользуемся литералами объекта и массива
```

```
const obj = new Object();
```

```
const arr = new Array();
```

typeof. Никакой магии

Простой пример:

```
console.log(typeof Object); 'function'  
  
console.log(typeof Array); 'function'
```

Object и *Array* — обычные функции. А что касательно классов?

```
class Aircraft {  
}  
  
console.log(typeof Aircraft); 'function'
```

typeof. Никакой магии

Результат конструкции *class* — обычная функция.

class удобен, так как в ES5 для работы в терминах ООП и такого же результата требовалось приложить больше усилий



Классы без *new*

Попытка **вызывать** полученную функцию без *new* приведёт нас к **ошибке**:

```
class Aircraft {  
}  
  
console.log(Aircraft());  
// Class constructor Aircraft cannot be invoked without 'new'
```


Функции-конструкторы

Вам уже знакомо понятие **функция-конструктор**. По сути, любая функция, если перед ней стоит оператор **new**, создаёт объект:

```
function Bobik() {  
    // эта функция вообще ничего не делает!  
}  
  
const bob = new Bobik();  
  
console.log(typeof bob); // object
```

Конструкция **class** в ES6 призвана **упростить создание объектов**, при этом не меняя принцип работы с этими объектами

.constructor

После вызова **конструкции** `new` у нового созданного объекта появляется автоматически свойство `constructor`:

```
const data = new Array();  
  
console.log(data.constructor); // [Function: Array]
```

Данное свойство ссылается на **функцию-конструктор**, породившую экземпляр

.constructor

А что будет у объектов, созданных через **конструкцию** *class*?

```
const data = new Aircraft();  
  
console.log(data.constructor); // [Function: Aircraft]
```

Всё точно также!



Методы

Как и свойства, у **классов можно предопределить методы** для всех создаваемых экземпляров этого типа.

Создадим метод, вычисляющий среднюю оценку спортсмена за выступление

Код на следующей странице



Методы

Обратите внимание, что использование `this` **внутри метода** позволяет обращаться к **текущему экземпляру класса**

Код на следующей странице

Методы

```
class Sportsman {
  constructor() {
    this.scores = [];
  }

  getAverageScore() {
    if (this.scores.length === 0) {
      return 0;
    }
    let sum = 0;

    // сумма оценок, делённая на их количество
    for (let rating of this.scores) {
      sum += rating;
    }
    return sum / this.scores.length;
  }
}
```

Продолжение кода на следующей странице

Методы

```
// добавляет новую оценку
rate(rating) {
    this.scores.push(rating);
}
}

const olga = new Sportsman();
olga.rate(10);
olga.rate(8);

console.log(olga.getAverageScore()); // 9
```

Вычисляемые методы

Благодаря ES6 у нас есть возможность **задавать методы класса**, которые заранее еще неизвестны:

```
const mySuperMethodName = 'getTrackName';

class MetallicaAlbum {
  [mySuperMethodName]() {
    return 'Enter Sandman';
  }
}

const album = new MetallicaAlbum();

console.log(album.getTrackName()); // 'Enter Sandman'
```




Геттеры и сеттеры объектов

У **методов объектов** есть возможность косить под свойства. Это позволяет перехватывать на ходу мысли программы. Например, данный код мимоходом устанавливает возраст человека, зная его дату рождения:

Код на следующей странице

Геттеры и сеттеры объектов

```
const person = {
  name: 'Владимир',

  /* это сеттер, пробел после set необходим
  единственный аргумент сеттера - значение, записываемое в него
  */

  set birthYear(year) {
    const date = new Date;
    this.age = date.getFullYear() - year;
  }
}

// вызываем метод, а обращаемся к свойству!
person.birthYear = 1980;

console.log(typeof person.age); 'number'
console.log(person.birthYear); // undefined
```



Геттеры и сеттеры объектов

В данном коде не показано, как узнать информацию о годе рождения, ведь мы даже **никуда не сохраняем эти данные**. Используя конструкцию вида:

```
this.birthYear = year
```

Мы просто ломаем наш код

Код на следующей странице

Геттеры и сеттеры объектов

```
const person = {  
  
  name: 'Владимир',  
  set birthYear(year) {  
    const date = new Date;  
    this.age = date.getFullYear() - year;  
  
    // Приведёт к переполнению стека. (Maximum call stack size exceeded)  
  
    this.birthYear = year;  
  }  
}  
  
person.birthYear = 1980;
```

В данном случае проблемной строкой является:

```
this.birthYear = year;
```

Геттеры и сеттеры объектов

Внутри сеттера строка опять обращается к сеттеру, который выполняет код, вновь доходит до указанной строки и вновь вызывает сеттер. **Из этого бесконечного круга нет выхода.**

Для того, чтобы иметь возможность **читать установленные значения** привычным способом, нам потребуется **геттер**



Геттеры и сеттеры объектов

```
const person = {
  name: 'Владимир',
  set birthYear(year) {
    const date = new Date;
    this.age = date.getFullYear() - year;
    this._birthYear = year; // (1)
  },

  // у геттера нет аргументов
  get birthYear() {
    return this._birthYear;
  }
}

person.birthYear = 1980; // сработал сеттер

console.log(typeof person.age); // 'number'
console.log(person.birthYear); // 1980, сработал геттер
```

В данном случае (1) мы использовали **новое свойство** `_birthYear` объекта.

Нижнее подчёркивание слева от названия ничего не значит, просто упрощает чтение и поиск

Геттеры и сеттеры объектов

Сеттеры и геттеры, заданные в классе, появляются во всех экземплярах:

```
class Person {  
  
    constructor(name, birthYear) {  
        this.name = name;  
        // работает сеттер  
        this.birthYear = birthYear;  
    }  
  
    set birthYear(year) {  
        const date = new Date;  
        this.age = date.getFullYear() - year;  
        this._birthYear = year;  
    }  
  
    get birthYear() {  
        return this._birthYear;  
    }  
}
```

*Продолжение кода на следующей
странице*

Геттеры и сеттеры объектов

Сеттеры и геттеры, заданные в классе, появляются во всех экземплярах.

```
const ivan = new Person('Иван', 1980);

// работает сеттер
ivan.birthYear = 1990;

// работает геттер
console.log(ivan.birthYear);
```


Наследование

Один из базовых принципов ООП — наследование.

Давайте представим, что есть Николай Петрович:

```
class Human {  
  
    constructor(name) {  
        this.name = name;  
    }  
}  
  
const human = new Human('Николай Петрович');
```

Наследование

Николай Петрович — мужчина:

```
class Man {  
  
    constructor() {  
        this.gender = "male";  
    }  
}  
  
const human = new Man();  
console.log(human.name); // undefined
```

Наследование

Очевидно, что все мужчины люди, и у всех есть имя. Можно ли как-то совместить эти два класса выше? **Можно!** С помощью **ключевого слова** *extends*:

```
class Human {  
  
    constructor(name) {  
        this.name = name;  
    }  
}  
  
class Man extends Human { }  
  
const human = new Man("Николай Петрович");  
  
console.log(human.name); // 'Николай Петрович'  
console.log(human.gender); // undefined
```

Наследование

Когда мы используем `extends`, мы говорим о том, что класс `Man` является наследником класса `Human`.

Обратите внимание, что у экземпляра присутствует свойство `name` с ожидаемым значением. Это значит, что **вызывался конструктор** `Human`, хотя мы и писали `new Man()`, а не `new Human()`

Наследование

В этом и **суть наследования**: квартира дедушки, заверенная нам по наследству, — наша квартира. В нашем примере экземпляру *Man* по наследству достался **конструктор** и, соответственно, имя



Наследование

Замечательно, но мы потеряли сведения о поле! Это произошло из-за того, что в **конструкторе** `Human` просто не указана информация о свойстве `gender`.

Вернём конструктор `Man`

Код на следующей странице

Наследование

```
class Human {  
  
    constructor(name) {  
        this.name = name;  
    }  
}  
  
class Man extends Human {  
  
    constructor() {  
        this.gender = "male";  
    }  
}  
  
const human = new Man("Николай Петрович");  
  
console.log(human.gender); // 'male'  
console.log(human.name); // undefined
```

Наследование

А теперь мы **потеряли** имя! Это произошло из-за того, что при создании экземпляра вызвался конструктор от `Man`, в котором нет информации об имени. Как же **совместить** всё воедино?





***super*: демо**

Для удобного вызова класса-родителя у нас есть **конструкция** *super*:

Код на следующей странице

super: демо

```
class Human {  
  
    constructor(name) {  
        this.name = name;  
    }  
}  
  
class Man extends Human {  
  
    constructor(name) {  
        // вызываем родительский конструктор (Human)  
        super(name);  
        this.gender = "male";  
    }  
}  
  
const human = new Man("Николай Петрович");  
  
console.log(human.gender); // 'male'  
console.log(human.name); // 'Николай Петрович', ура!
```



super до *this*

Мы обязаны пользоваться **конструкцией** *super* до первого обращения к *this*,
иначе нас ждёт ошибка

Код на следующей странице

super до *this*

```
class Human {  
  
    constructor(name) {  
        this.name = name;  
    }  
}  
  
class Man extends Human {  
  
    constructor(name) {  
        this.gender = "male";  
        // super должен быть до первого this  
        super(name);  
    }  
}  
  
// Ошибка: Must call super constructor in derived class  
// before accessing 'this' or returning from derived constructor  
  
const human = new Man("Николай Петрович");
```

Наследование методов

Точно как и со свойствами и конструкторами, классы **могут наследовать** и методы:

```
class TextMessage {  
  
    read() {  
        console.log("Вам письмо, танцуйте!");  
    }  
}  
  
class SMS extends TextMessage { }  
  
const textMsg = new TextMessage();  
const msg = new SMS();  
  
textMsg.read(); // 'Вам письмо, танцуйте!'  
msg.read(); // 'Вам письмо, танцуйте!'
```

Собственные методы

Методы, созданные в расширенном классе, **недоступны для родителя**:

```
// форма на сайте
class SiteForm { }

// форма обратной связи
class CallbackForm extends SiteForm {

  onSend() {
    console.log("Спасибо за заявку! Мы свяжемся с вами в ближайшее время");
  }
}

const form = new SiteForm();
const callbackForm = new CallbackForm();

console.log(typeof form.onSend); // undefined
console.log(typeof callbackForm.onSend); // function
```



Полиморфизм

Второй принцип, который есть в ООП — **полиморфизм**.

Собака и улитка передвигаются, но каждый делает это по-разному. Так и **объекты** могут иметь одни и те же методы, но реализация этих методов может отличаться

Код на следующей странице

Полиморфизм

```
class VideoItem {  
  constructor(title) {  
    this.title = title;  
  }  
  play() {  
    console.log(`Начинаю воспроизводить видео ${this.title}`);  
  }  
}
```

Первая часть кода

Полиморфизм

```
// видео с рекламой
class AdVideoItem extends VideoItem {
  play() {
    alert("Исландский морж улетел в космос! Кликай сюда!");
    console.log(`Начинаю воспроизводить видео ${this.title}`);
  }
}

const video = new VideoItem("Как разбогатеть на чтении!");
const adsVideo = new AdVideoItem("Ванга рассказала Киркорову про ЭТО!");

video.play(); // 'Начинаю воспроизводить видео Как разбогатеть на чтении!'
adsVideo.play(); // 'Исландский морж улетел в космос! Кликай сюда!'
```

Вторая часть кода

Полиморфизм

Оба экземпляра, несмотря на различие в реализации метода *play*, обладают свойством *title*





super в полиморфизме

С помощью *super* можно **обращаться** к методам родительского класса:

Код на следующей странице

super в полиморфизме

```
class VideoItem {
  constructor(title) {
    this.title = title;
  }

  play() {
    console.log(`Начинаю воспроизводить видео ${this.title}`);
  }
}

class AdsVideoItem extends VideoItem {
  play() {
    alert("Исландский морж улетел в космос! Кликай сюда!");
    super.play(); // Вызываем play у VideoItem
  }
}
```

Продолжение кода на следующей странице

super в полиморфизме

```
const video = new VideoItem("Как разбогатеть на чтении!");  
const adsVideo = new AdsVideoItem("Ванга рассказала Киркорову про ЭТО!");  
  
// тот же результат  
video.play();  
adsVideo.play();
```

Многоуровневое наследование: демо

Многоуровневое наследование работает также, как и в случае с двумя классами:

```
class A {  
  
    // возвращает случайное число. Кстати, это пригодится в ДЗ!  
    getRandomNumber() {  
        return Math.random();  
    }  
}  
  
class B extends A { }  
class C extends A { }  
  
const bobik = new C();  
  
bobik.getRandomNumber(); // случайное число в диапазоне 0 и 1
```

Статические методы

Так как **класс** — это **обычная функция**, а функция в JS представлена объектом, у этого объекта можно определить методы. Такие методы в терминологии ES6 называются **статическими**:

```
class Text {  
  
    static isText(str) {  
        return typeof str === "string";  
    }  
}  
  
Text.isText("В чём смысл жизни?"); // true  
Text.isText(42); // false
```

Статические методы отсутствуют в экземплярах

```
class Text {  
  
    static isText(str) {  
        return typeof str === "string";  
    }  
}  
  
const text = new Text();  
  
// статические методы отсутствуют в экземплярах  
console.log(text.isText); // undefined
```


Статические методы отсутствуют в экземплярах

Примеры статических методов в самом JS:

```
Array.isArray(null);
```

```
Array.of([345, 7]);
```


```
Array.from(4);
```

```
Object.keys({ hello: "world" });
```

this в статических методах

this в статических методах указывает на сам класс *или функцию-конструктор*, так как статические методы вызываются **вне экземпляра**:

```
class Test {  
  
    static showThis() {  
        console.log(this);  
    }  
}  
  
Test.showThis(); // [Function: Test]
```



Нововведения последних версий ES

Статические свойства

Внимание! Так как это нововведения, то **не во всех браузерах** они могут работать:

```
class Text {  
    static TYPE_TEXT = "text";  
    static TYPE_EMAIL = "email";  
    static TYPE_PHONE = "phone";  
}  
  
console.log(Text.TYPE_TEXT); // 'text'
```

Статические свойства

В **старых браузерах** для реализации такого функционала нам нужно написать:

```
class Text { }

Text.TYPE_TEXT = "text";
Text.TYPE_EMAIL = "email";
Text.TYPE_PHONE = "phone";

console.log(Text.TYPE_TEXT); // 'text'
```



Приватные свойства: демо

В современном JS была добавлена возможность **приватных полей**, к которым можно обратиться **только внутри класса**

Код на следующей странице.

Более подробно о полях класса:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Classes/Private_class_fields

Приватные свойства: демо

```
class Cat {  
  
    #health;  
  
    constructor() {  
        this.#health = 9;  
        this.#hungry = 0;  
        //SyntaxError: Private field '#hungry' must be declared in an enclosing class  
    }  
  
    getHealth() {  
        return this.#health;  
    }  
}  
  
const kitty = new Cat();  
  
console.log(kitty.#health);  
// SyntaxError: Private field '#health' must be declared in an enclosing class  
console.log(kitty.getHealth()); // 9
```



Итоги



Чему мы научились?

- **Разобрались** с концепцией ООП в JS
- **Изучили** тенденции ES6
- **Узнали** преимущества создания объектов с помощью классов
- **Научились** создавать шаблонные методы и свойства для всех экземпляров класса
- **Познакомились** с принципами наследования и полиморфизма
- **Узнали**, что было добавлено в последних спецификациях ES

Домашнее задание

Давайте посмотрим ваше [домашнее задание](#):

- **вопросы** по домашней работе задаём в группе Slack
- задачи можно сдавать **по частям**
- зачёт по домашней работе проставляется после того, как приняты **все обязательные задачи**

**Задавайте вопросы и
пишите отзыв о лекции!**

Владимир Чебукин