



ООП В JS (ES6)



ВЛАДИМИР ЧЕБУКИН



ВЛАДИМИР ЧЕБУКИН

Веб-разработчик



vovachebr@mail.ru



fb.me/vovachebr



[@User123423](https://t.me/@User123423)



ПЛАН ЗАНЯТИЯ

1. Классы
2. Свойства
3. Конструктор, методы, статические методы
4. Наследование и `super`
5. Нововведения последних версий ES



ООП

ООП — методология разработки программ, в которой все важные вещи представляются объектами. Каждый объект построен по определенным правилам, которые называют классом. Классы основываются друг на друге, что называют наследованием.

4 КИТА

Четыре принципа, поверх которых строятся объектно-ориентированные приложения:

1. Абстракция — рассмотрения объекта реального мира в контексте конкретной задачи.
2. Инкапсуляция — сокрытие внутренней реализации.
3. Наследование — передача характеристик одних объектов другим через отношение «является» (кот является животным).
4. Полиморфизм — возможность работать с конкретной структурой данных, будто с абстрактной.



ПРИМЕР ОБЪЕКТНО-ОРИЕНТИРОВАННОГО РЕШЕНИЯ (ДЕМО)

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



ПРЕИМУЩЕСТВА ООП-СТИЛЯ

1. Идеально подходит для большого количества типовых объектов.
2. Позволяет удобно делить сложные конструкции на мелкие составляющие.
3. Упрощает работу со внутренним состоянием.

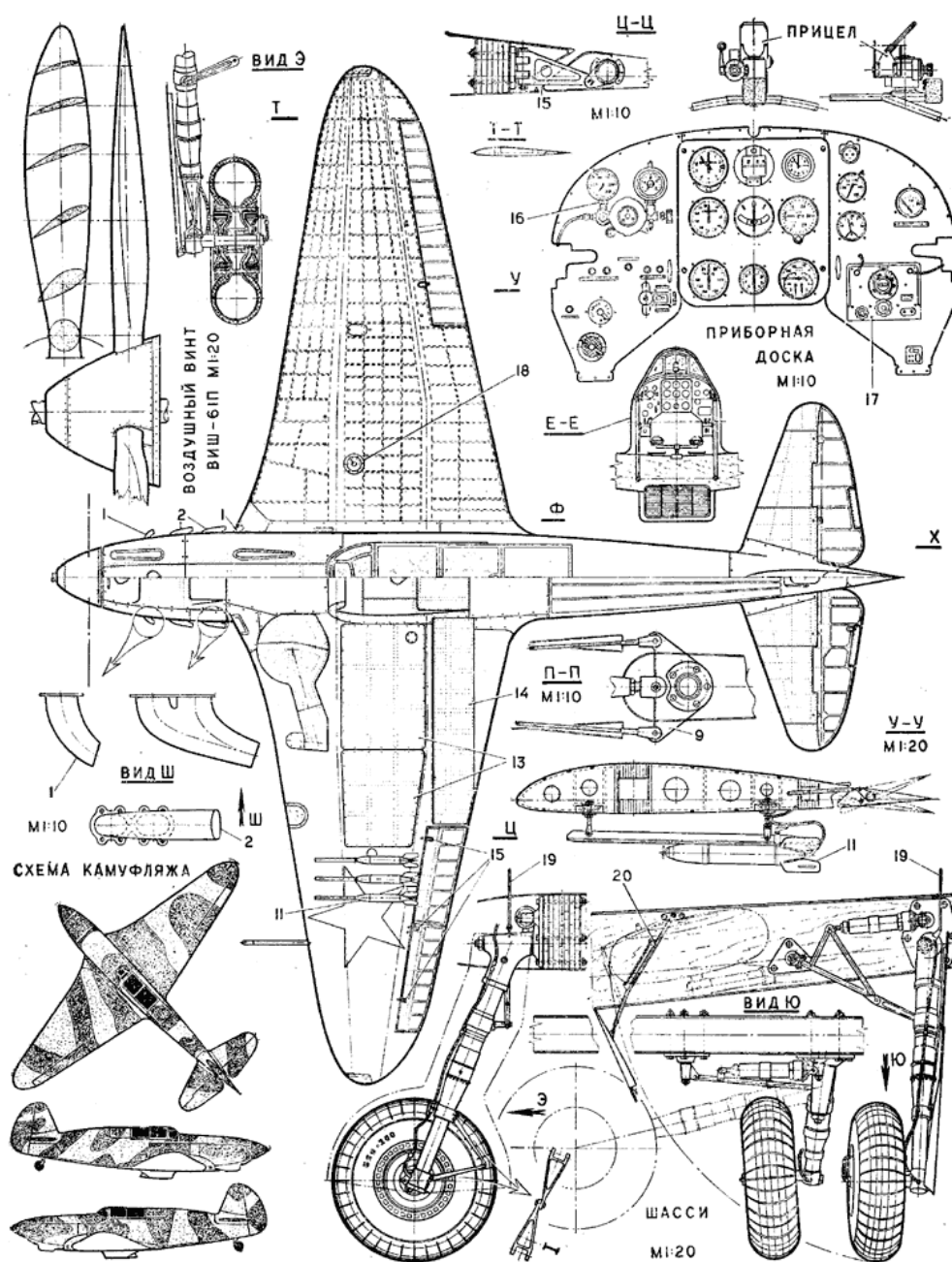
ООП стиль — это не обязательно меньше строчек кода в сравнении с другими парадигмами, но зачастую лучшее понимание и чтение кода.

class

В ES6 добавилась новая конструкция — `class`.

Класс представляет макет, по которому будет создан конкретный объект. Точно также, как по чертежу самолёта делают самолёт.

```
class Aircraft {  
}
```

new

`new` создаёт по «чертежу» класса экземпляр типа. Например:

```
1  class BMW {  
2  }  
3  
4  const bmw1 = new BMW();  
5  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 разные, со своей «душой».

Конструкция `new` всегда возвращает объект.

ТЕРМИНОЛОГИЯ: ТИПЫ И КЛАССЫ

Так как в JavaScript конструкция `class` — просто удобное сокращение существовавших ранее подходов.

```
1  class BMW {  
2  }  
3  
4  function BMW {  
5  }
```

В сторонних источниках говорят о том, что есть «тип BMW». Некоторые пользуются термином «класс BMW». Используйте любой термин.

ЭКЗЕМПЛЯРЫ – ОБЫЧНЫЕ ОБЪЕКТЫ

Экземпляры типов, заданных конструкцией `class` – обычные объекты. Они обладают теми же свойствами, что и обычный объект. Им также можно задавать свойства и методы:

```
1  const obj = {};  
2  
3  obj.title = 'Я - обычный объект!';  
4  obj.showTitle = function() {  
5      console.log(this.title);  
6  }  
7  
8  class SuperObject {  
9  }  
10  
11  const superObj = new SuperObject();  
12  // аналогично  
13  superObj.title = 'Супер';  
14  superObj.showTitle = function() {  
15      console.log(this.title);  
16  }
```

КОНСТРУКТОР КЛАССА

Для гибкой настройки объектов, им можно передавать начальные параметры. Например, было бы здорово задать название книги для класса `Book`:

```
1 class Book {  
2 }  
3  
4 const book = new Book('Понедельник начинается в субботу');
```

Но как воспользоваться этим значением?

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

```
1 class Book {  
2   constructor(name) {  
3     console.log(`Вы хотите создать книгу с названием «${name}»`);  
4   }  
5 }  
6  
7 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.

```
1 class Book {  
2   constructor(name) {  
3     this.name = name;  
4   }  
5 }  
6  
7 const book = new Book('Понедельник начинается в субботу');  
8 console.log(book.name); // 'Понедельник начинается в субботу'
```

Вот вариант для того, чтобы упорядочить простую телефонную книгу:

```
1  class Person {
2      constructor(firstName, lastName, phone) {
3          this.firstName = firstName;
4          this.lastName = lastName;
5          this.phone = phone;
6      }
7  }
8  const phonebook = [
9      new Person('Владислав', 'Иванов', '+74993412233'),
10     new Person('Леонида', 'Петрова', '+74993412232'),
11 ];
```


МАГИЯ КОНСТРУКТОРА

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

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

и

```
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  const oleg = new Male('Олег', 'Иванов');
10 const ivan = new Male('Иван', 'Широков');
11 const nikita = new Male('Никита', 'Огурцов');
```

Таким образом, нам удалось без особого труда добавить 2 дополнительных свойства, которые будут присутствовать абсолютно во всех экземплярах объекта.

`typeof`. НИКАКОЙ МАГИИ

Классы в JS — так называемый синтаксический сахар. Эта конструкция, пришедшая с ES6, делает жизнь разработчика удобнее, не более.

Давайте посмотрим, что из себя представляет `Object` и `codeArray` в конструкциях

```
// тут мы не пользуемся литералами объекта и массива  
const obj = new Object();  
const arr = new Array();
```

Проводим простой пример:

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

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

```
1  class Aircraft {  
2  }  
3  
4  console.log(typeof Aircraft); 'function'
```

Результат конструкции `class` — обычная функция. `class` просто удобен, так как в ES5 для работы в терминах ООП и такого же результата требовалось осуществить больше усилий.

КЛАССЫ БЕЗ `new`

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

```
1 class Aircraft {  
2 }  
3  
4 console.log(Aircraft());  
5 // Class constructor Aircraft cannot be invoked without 'new'
```

ФУНКЦИИ-КОНСТРУКТОРЫ

В JS существует такое понятие — функция-конструктор. Это та функция, которая создаёт объект. По сути, любая функция, имеющая перед этим `new`, создаёт объект:

```
1 function Bobik() {  
2   // эта функция вообще ничего не делает!  
3 }  
4  
5 const bob = new Bobik();  
6  
7 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. А что будет у объектов, созданных через конструкцию `class`?

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

Всё точно так же!



МЕТОДЫ

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

Например, создадим метод, вычисляющий среднюю оценку спортсмена за выступление.


```
1 class Sportsman {
2   constructor() {
3     this.scores = [];
4   }
5   getAverageScore() {
6     if (this.scores.length === 0) {
7       return 0;
8     }
9     let sum = 0;
10    // сумма оценок, делённая на их количество
11    for (let rating of this.scores) {
12      sum += rating;
13    }
14    return sum / this.scores.length;
15  }
16  // добавляет новую оценку
17  rate(rating) {
18    this.scores.push(rating)
19  }
20 }
21
22 const olga = new Sportsman();
23 olga.rate(10);
24 olga.rate(8);
25 console.log(olga.getAverageScore()); // 9
```

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

ВЫЧИСЛЯЕМЫЕ МЕТОДЫ

Благодаря ES6, у нас есть возможность задания методов класса, которые заранее ещё неизвестны:

```
1  const mySuperMethodName = 'getTrackName';
2
3  class MetallicaAlbum {
4      [mySuperMethodName]() {
5          return 'Enter Sandman';
6      }
7  }
8
9  const album = new MetallicaAlbum();
10 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:

```
1  const person = {
2    name: 'Владимир',
3
4    /*
5     это сеттер, пробел после set необходим
6     единственный аргумент сеттера – значение, записываемое в него
7    */
8    set birthYear(year) {
9      const date = new Date();
10     this.age = date.getFullYear() - year;
11   }
12 }
13
14 // вызываем метод, а обращаемся к свойству!
15 person.birthYear = 1980;
16 console.log(typeof person.age); 'number'
17 console.log(person.birthYear); // undefined
```

В данном коде не показано, как узнать информацию о годе рождения, ведь мы даже никуда не сохраняем эту информацию. Если использовать конструкцию вида

Если использовать конструкцию вида `this.birthYear = year`, то мы просто сломаем наш код:

```
1  const person = {
2    name: 'Владимир',
3    set birthYear(year) {
4      const date = new Date();
5      this.age = date.getFullYear() - year;
6      /*
7       * Приведёт к переполнению стека.
8       * (Maximum call stack size exceeded)
9       */
10     this.birthYear = year;
11   }
12 }
13
14 person.birthYear = 1980;
```



В данном случае проблемной строкой является:

```
this.birthYear = year
```

Внутри сеттера она 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.

```
1  const person = {
2    name: 'Владимир',
3
4    set birthYear(year) {
5      const date = new Date();
6      this.age = date.getFullYear() - year;
7      /*
8       ничего не значит и не вносит сакральности
9       Мы просто хотим сохранить сеттер
10      */
11      this._birthYear = year;
12    },
13
14    // у геттера нет аргументов
15    get birthYear() {
16      return this._birthYear;
17    }
18  }
19
20  person.birthYear = 1980; // сработал сеттер
21  console.log(typeof person.age); 'number'
22  console.log(person.birthYear); // 1980, сработал геттер
```

В данном случае мы использовали новое свойство `_birthYear` объекта. Нижнее подчёркивание слева от названия ничего не значит, просто упрощает чтение и поиск.

ГЕТТЕРЫ И СЕТТЕРЫ КЛАССОВ

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

```
1 class Person {
2   constructor(name, birthYear) {
3     this.name = name;
4     // сработает сеттер
5     this.birthYear = birthYear;
6   }
7   set birthYear(year) {
8     const date = new Date();
9     this.age = date.getFullYear() - year;
10    this._birthYear = year;
11  },
12  get birthYear() {
13    return this._birthYear;
14  }
15 }
16
17 const ivan = new Person('Иван', 1980);
18 // сработает сеттер
19 ivan.birthYear = 1990;
20 // сработает геттер
21 console.log(ivan.birthYear);
```


НАСЛЕДОВАНИЕ

Один из базовых принципов ООП: наследование.

Давайте представим, что есть Николай Петрович:

```
1  class Human {  
2      constructor(name) {  
3          this.name = name;  
4      }  
5  }  
6  
7  const human = new Human( 'Николай Петрович' );
```

Николай Петрович — мужчина.

```
1 class Man {  
2     constructor() {  
3         this.gender = 'male';  
4     }  
5 }  
6  
7 const human = new Man();  
8 console.log(human.name); // undefined
```

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`:

```
1 class Human {  
2     constructor(name) {  
3         this.name = name;  
4     }  
5 }  
6  
7 class Man extends Human {  
8 }  
9  
10 const human = new Man('Николай Петрович');  
11 console.log(human.name); // 'Николай Петрович'  
12 console.log(human.gender); // undefined
```

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()`.

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.

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`:

```
1  class Human {
2      constructor(name) {
3          this.name = name;
4      }
5  }
6
7  class Man extends Human {
8      constructor() {
9          this.gender = 'male';
10     }
11 }
12
13 const human = new Man('Николай Петрович');
14 console.log(human.gender); // 'male'
15 console.log(human.name); // undefined
```

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`:

```
1  class Human {
2      constructor(name) {
3          this.name = name;
4      }
5  }
6
7  class Man extends Human {
8      constructor(name) {
9          // вызываем родительский конструктор (Human)
10         super(name)
11         this.gender = 'male';
12     }
13 }
14
15 const human = new Man('Николай Петрович');
16 console.log(human.gender); // 'male'
17 console.log(human.name); // 'Николай Петрович', ура!
```

super ДО this

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

```
1  class Human {
2      constructor(name) {
3          this.name = name;
4      }
5  }
6
7  class Man extends Human {
8      constructor(name) {
9          this.gender = 'male';
10         // super должен быть до первого this
11         super(name);
12     }
13 }
14
15 /*
16     Ошибка: Must call super constructor in derived class
17     before accessing 'this' or returning from derived constructor
18 */
19 const human = new Man('Николай Петрович');
```

НАСЛЕДОВАНИЕ МЕТОДОВ

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

```
1 class TextMessage {  
2     read() {  
3         console.log('Вам письмо, танцуйте!');  
4     }  
5 }  
6  
7 class SMS extends TextMessage {  
8  
9 }  
10  
11 const textMsg = new TextMessage();  
12 const msg = new SMS();  
13  
14 textMsg.read(); // 'Вам письмо, танцуйте!'  
15 msg.read(); // 'Вам письмо, танцуйте!'
```


СОБСТВЕННЫЕ МЕТОДЫ

Методы, созданные в расширенном классе, недоступны для родителя:

```
1 // форма на сайте
2 class SiteForm {
3 }
4
5 // форма обратной связи
6 class CallbackForm extends SiteForm {
7     onSend() {
8         console.log('Спасибо за заявку! Мы свяжемся с вами в ближайшее время');
9     }
10 }
11
12 const form = new SiteForm();
13 const callbackForm = new CallbackForm();
14
15 console.log(typeof form.onSend); // undefined
16 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.

```
1 class VideoItem {
2   constructor(title) {
3     this.title = title;
4   }
5
6   play() {
7     console.log('Начинаю воспроизводить видео ${this.title}');
8   }
9 }
10
11 // видео с рекламой
12 class AdsVideoItem extends VideoItem {
13   play() {
14     alert('Исландский морж улетел в космос! Кликай сюда!');
15     console.log('Начинаю воспроизводить видео ${this.title}');
16   }
17 }
18
19 const video = new VideoItem('Как разбогатеть на чтении!');
20 const adsVideo = new AdsVideoItem('Ванга рассказала Киркорову про ЭТО!');
21
22 video.play(); // 'Начинаю воспроизводить видео Как разбогатеть на чтении!'
23 adsVideo.play(); // 'Исландский морж улетел в космос! Кликай сюда!' и ...
```

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

super В ПОЛИМОРФИЗМЕ

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

```
1 class VideoItem {
2   constructor(title) {
3     this.title = title;
4   }
5
6   play() {
7     console.log(`Начинаю воспроизводить видео ${this.title}`);
8   }
9 }
10
11 class AdVideoItem extends VideoItem {
12   play() {
13     alert('Исландский морж улетел в космос! Кликай сюда!');
14     // Вызываем play у VideoItem
15     super.play();
16   }
17 }
18
19 const video = new VideoItem('Как разбогатеть на чтении!');
20 const adsVideo = new AdVideoItem('Ванга рассказала Киркорову про ЭТО!');
21
22 // тот же результат
23
24 video.play();
25 adsVideo.play();
```

МНОГОУРОВНЕВОЕ НАСЛЕДОВАНИЕ (ДЕМО)

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

```
1  class A {  
2    // возвращает случайное число. Кстати, это пригодится в ДЗ!  
3    getRandomNumber() {  
4      return Math.random();  
5    }  
6  }  
7  
8  class B extends A {  
9  }  
10  
11  class C extends B {  
12  }  
13  
14  const bobik = new C();  
15  
16  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:

```
1 class Text {  
2   static isText(str) {  
3     return typeof str === 'string';  
4   }  
5 }  
6  
7 Text.isText('В чём смысл жизни?'); // true  
8 Text.isText(42); // false
```

СТАТИЧЕСКИЕ МЕТОДЫ ОТСУТСТВУЮТ В ЭКЗЕМПЛЯРАХ!

```
1 class Text {  
2     static isText(str) {  
3         return typeof str === 'string';  
4     }  
5 }  
6 const text = new Text();  
7 // статические методы отсутствуют в экземплярах  
8 console.log(text.isText); // undefined
```

Примеры статических методов в самом JS:

```
1 Array.isArray(null);  
2 Array.of([345, 7]);  
3 Array.from(4);  
4  
5 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:

```
1 class Test {  
2     static showThis() {  
3         console.log(this);  
4     }  
5 }  
6  
7 Test.showThis(); // [Function: Test]
```

СТАТИЧЕСКИЕ СВОЙСТВА

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

```
1 class Text {  
2     static TYPE_TEXT = 'text';  
3     static TYPE_EMAIL = 'email';  
4     static TYPE_PHONE = 'phone';  
5 }  
6  
7 console.log(Text.TYPE_TEXT); // 'text'
```

В старых браузерах для реализации такого функционала нам нужно написать:

```
1 class Text {  
2 }  
3 Text.TYPE_TEXT = 'text';  
4 Text.TYPE_EMAIL = 'email';  
5 Text.TYPE_PHONE = 'phone';  
6  
7 console.log(Text.TYPE_TEXT); // 'text'
```

ПРИВАТНЫЕ СВОЙСТВА (ДЕМО)

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

```
1 class Cat {
2   #health;
3
4   constructor() {
5     this.#health = 9;
6     this.#hungry = 0;
7     // SyntaxError: Private field '#hungry' must be declared in an enclosing class
8   }
9
10  getHealth(){
11    return this.#health;
12  }
13 }
14
15 const kitty = new Cat();
16 console.log(kitty.#health);
17 // SyntaxError: Private field '#health' must be declared in an enclosing class
18 console.log(kitty.getHealth());
19 // 9
```

[Более подробно о полях класса](#)

ЧЕМУ МЫ НАУЧИЛИСЬ?

1. Разобрались с концепцией ООП в JS;
2. Изучили тенденции ES6;
3. Узнали преимущества создания объектов с помощью классов;
4. Научились создавать шаблонные методы и свойства для всех экземпляров класса;
5. Познакомились с принципами наследования и полиморфизма;
6. Узнали, что было добавлено в последних спецификациях ES.



ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты все задачи.



Спасибо за внимание!
Время задавать вопросы

ВЛАДИМИР ЧЕБУКИН



vovachebr@mail.ru



fb.me/vovachebr



[@User123423](https://t.me/@User123423)