НЕТОЛОГИЯ

ООП В JS (ES6)

ВЛАДИМИР ЧЕБУКИН

ВЛАДИМИР ЧЕБУКИН

Frontend-разработчик

vovachebr@mail.ru fb.me/vovachebr @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 структорой 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.

(На двух слайдах)

```
class Cart {
    constructor() {
    // внутреннее хранилище
    this.items = []
    }

    find(product) {
    return this.items.find(function(cartItem ) {
      return cartItem.product.id === product.id;
    });
}

    add(product) {
    const item = this.find(product);

    if (item) {
      item.quantity++;
      return;
    }

    this.items.push({
      product,
      quantity: 1
      });
    }
}
```

(продолжение)

```
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);
```

ПРЕИМУЩЕСТВА ООП-СТИЛЯ

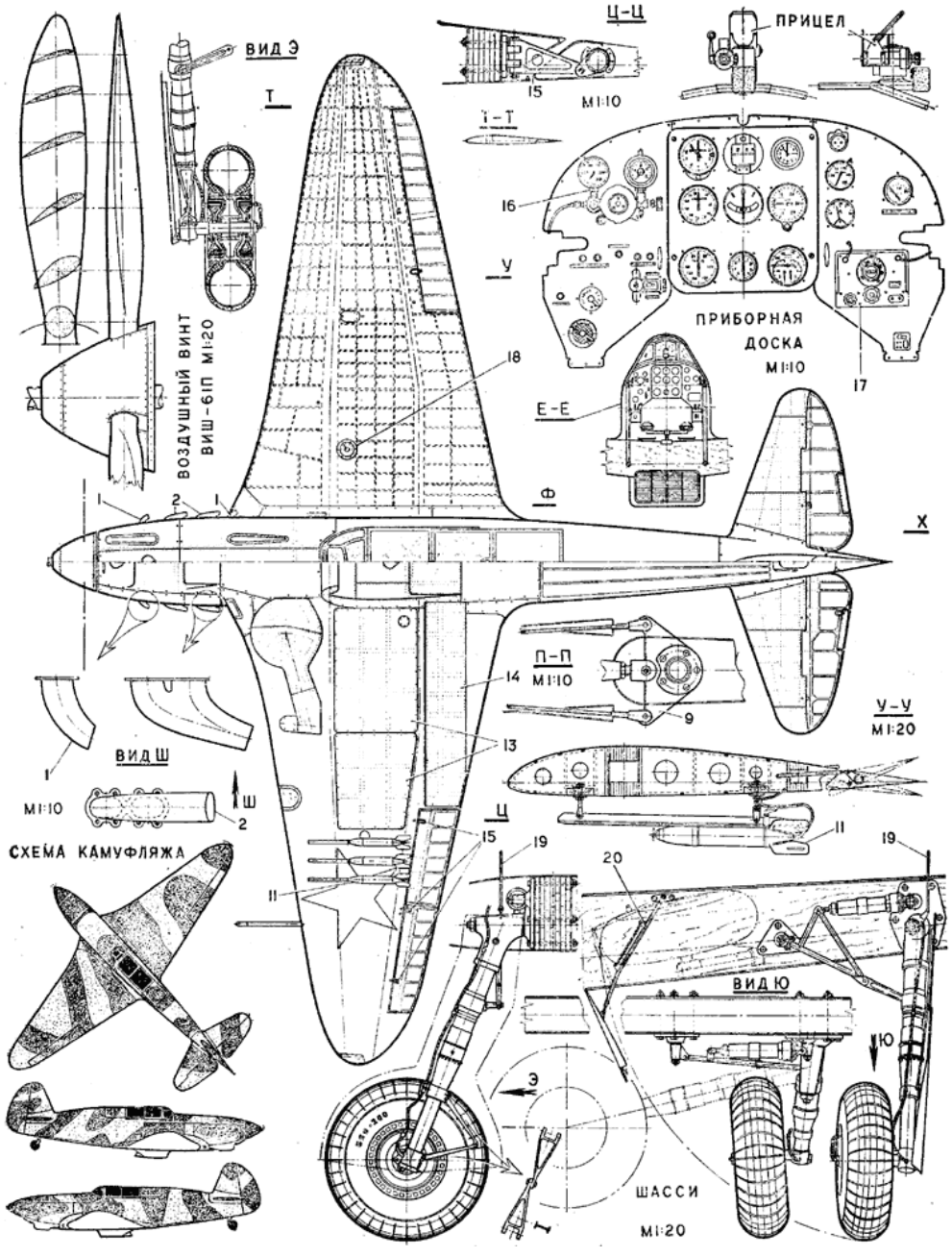
1. Идеально подходит для большого количества типовых объектов.
2. Позволяет удобно делить сложные конструкции на мелкие составляющие.
3. Упрощает работу со внутренним состоянием.

`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 {
}
```


ВИД Э
Т
ВОЗДУШНЫЙ ВИНТ
ВИШ-61П МИ-20
Ц-Ц
15
МИ-10
ПРИЦЕЛ
Т-Т
16
У
ПРИБОРНАЯ
ДОСКА
МИ-10
17
18
Е-Е
Ф
Х
П-П
МИ-10
14
9
У-У
МИ-20
13
ВИД Ш
1
МИ-10
Ш
2
Ц
15
19
11
20
СХЕМА КАМУФЛЯЖА
ВИД Ю
Ю
Э
ШАССИ
МИ-20

new

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

```
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 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
6  }
```

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:

```
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?

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'
```

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:

```
const oleg = {
  name: 'Олег',
  lastName: 'Иванов',
  gender: 'male',
  type: 'human'
}
const ivan = {
  name: 'Иван',
  lastName: 'Широков',
  gender: 'male',
  type: 'human'
}
const nikita = {
  name: 'Никита',
  lastName: 'Огурцов',
  gender: 'male',
  type: 'human'
}
```

и

```
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');
```

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?

```
class Aircraft {
}

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:

```
class Aircraft {
}

console.log( Aircraft());
// 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:

```
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. 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.

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

const olga = new Sportsman();

olga.rate(10);
olga.rate(8);

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:

```
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:

```
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
```

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:

```
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
```

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.

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

  set birthYear(year) {
    const date = new Date();
    this.age = date.getFullYear() - year;
    /*
    ничего не значит и не вносит сакральности
    Мы просто хотим сохранить сеттер
    */
    this._birthYear = year;
  },

  // 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 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:

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

const ivan = new Person('Иван', 1980);
// сработает сеттер
ivan.birthYear = 1990;
// сработает 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');
```

Николай Петрович — мужчина.

```
class Man {
  constructor() {
    this.gender = 'male';
  }
}

const human = new Man();
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`:

```
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
```

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`:

```
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
```

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`:

```
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:

```
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.

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

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

// видео с рекламой
class AdsVideoItem 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 AdsVideoItem('Ванга рассказала Киркорову про ЭТО!');

video.play();   // 'Начинаю воспроизводить видео Как разбогатеть на чтении!'
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:

```
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!');
    // Вызываем play у VideoItem
    super.play();
  }
}

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 так же, 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
```

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]
```

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'
```

В старых брузерах 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**.

```
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
```

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**.

НЕТОЛОГИЯ

Спасибо за внимание!

Время задавать вопросы ☺

ВЛАДИМИР ЧЕБУКИН


vovachebr@mail.ru


fb.me/vovachebr


@User123423