

# КЛАССЫ И НАСЛЕДОВАНИЕ

ЛЕКТОР / ДОЛЖНОСТЬ

# ЛЕКТОР

ДОЛЖНОСТЬ





# ПЛАН ЗАНЯТИЯ

1. [Иерархия наследования](#)
2. [Классы](#)
3. [Конструкторы](#)
4. [Наследование](#)
5. [Статические методы](#)

# ПОВТОРЕНИЕ

На предыдущих лекциях мы с вами рассмотрели:

- цепочки прототипов и свойство `__proto__`
- функции конструкторы и свойство `prototype`



# ИЕРАРХИЯ НАСЛЕДОВАНИЯ



# НАСЛЕДОВАНИЕ

Наследование (Inheritance) — механизм, подразумевающий переиспользование свойств «родительского объекта» или «родительского класса» в дочернем.

При этом выделяют два ключевых вида наследования:

- на базе прототипов (на базе объектов)
- на базе классов



# НАСЛЕДОВАНИЕ В JS

Наследование в JS строится на базе цепочки прототипов (уже рассмотренного нами механизма). Т.е. любое свойство сначала ищется в самом объекте, потом в прототипе объекта, потом в прототипе прототипа и т.д.



# ЗАДАЧА

Перед нами встала задача организовать веб-мессенджер.

В базовой версии он должен позволять обмениваться сообщениями только пользователям, зарегистрированным в нашей системе.

А затем мы хотим подготовить специализированные версии, которые позволят общаться с пользователями других мессенджеров, например, Viber.





# OLD STYLE

# ФУНКЦИЯ КОНСТРУКТОР

```
1 function Messenger(name) {  
2   this.name = name;  
3 }
```

# МЕТОД В ПРОТОТИПЕ

```
1 Messenger.prototype = {  
2   send(recipient, msg) {  
3     // TODO: send text message  
4   },  
5 };
```



# СПЕЦИАЛИЗАЦИЯ

Первое, чего мы хотим добиться, — чтобы у каждого специализированного мессенджера были в наличии все те же свойства, что есть и в базовом.

# НАСЛЕДОВАНИЕ СВОЙСТВ

```
1  function Messenger(name) {  
2    this.name = name;  
3  }  
4  
5  function MultiMessenger(name) {  
6    Messenger.call(this, name); // <-  
7  }  
8  MultiMessenger.prototype = Object.create(Messenger.prototype);  
9  MultiMessenger.prototype.constructor = MultiMessenger;  
10  
11  const viber = new MultiMessenger('Viber');  
12  console.log(viber.name); // Viber
```

---

## OBJECT.CREATE

Метод, позволяющий создать новый объект с установленным объектом прототипа. Фактически, мы в свойство `prototype` нашей функции конструктора прописываем объект, у которого в прототипе будет свойство `prototype` из `Messenger`.



# СПЕЦИАЛИЗАЦИЯ

Второе, — нужно иметь возможность добавлять собственные свойства.

# ДОБАВЛЕНИЕ СВОЙСТВ

```
1 function Messenger(name) {  
2   this.name = name;  
3 }  
4  
5 function MultiMessenger(name, logo) {  
6   Messenger.call(this, name);  
7   this.logo = logo; // <-  
8 }  
9 MultiMessenger.prototype = Object.create(Messenger.prototype);  
10 MultiMessenger.prototype.constructor = MultiMessenger;  
11  
12 const viber = new MultiMessenger('Viber', 'V');  
13 console.log(viber.name); // Viber  
14 console.log(viber.logo); // V
```





# МЕТОДЫ

Посмотрим, что с методами.

# НАСЛЕДОВАНИЕ МЕТОДОВ

```
1  function Messenger() { ... }
2  Messenger.prototype = {
3    send(recipient, msg) {
4      // TODO: send text message
5    },
6  };
7
8  function MultiMessenger() { ... }
9  MultiMessenger.prototype = Object.create(Messenger.prototype);
10 MultiMessenger.prototype.constructor = MultiMessenger;
11
12 const viber = MultiMessenger();
13 viber.send('...');
```



## МЕТОДЫ

Теперь нужно добавить свои — так, чтобы можно было посылать сообщения пользователям Viber.

# ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДА

Можем ли мы переопределить метод `send` в `MultiMessenger`?

```
1 function Messenger() { ... }
2 Messenger.prototype = {
3   send(recipient, msg) {
4     // TODO: send text message
5   },
6 };
7
8 function MultiMessenger() { ... }
9 MultiMessenger.prototype = Object.create(Messenger.prototype);
10 MultiMessenger.prototype.send = function(recipient, msg) { ... };
11 MultiMessenger.prototype.constructor = MultiMessenger;
12
13 const viber = MultiMessenger();
14 viber.send('...');
```



# МЕТОДЫ

Стоп, но тогда мы уже не сможем отправлять сообщения пользователям нашего сервиса. Как это исправить?

# ВЫЗОВ РОДИТЕЛЬСКОГО МЕТОДА

```
1 function Messenger() { ... }
2 Messenger.prototype = {
3   send(recipient, msg) {
4     // TODO: send text message
5   },
6 };
7
8 function MultiMessenger() { ... }
9 MultiMessenger.prototype = Object.create(Messenger.prototype);
10 MultiMessenger.prototype.send = function(recipient, msg) {
11   if (<recipient is from our service>) {
12     Messenger.prototype.send.call(this, recipient, msg);
13     return;
14   }
15
16   // esle send via Viber
17 };
18 MultiMessenger.prototype.constructor = MultiMessenger;
19
20 const viber = MultiMessenger();
21 viber.send('...');
```

---

# ES6

Манипуляция прототипами позволяет добиться нужного уровня гибкости, но в большинстве случаев является избыточной.

ES6 принёс нам ключевые слова `class` и `extends`, позволяющие использовать аналогичные другим языкам конструкции для создания функций-конструкторов и цепочек прототипов.



# КЛАССЫ





# CLASS

Удобная форма или «синтаксический сахар», позволяющий объединить создание функции-конструктора и добавление функций в прототипы.

# CLASS

```
1  class Messenger {  
2      constructor(name) { // Аналог функции конструктора  
3          this.name = name;  
4      }  
5  
6      send(recipient, msg) { // Аналог .prototype.send  
7  
8      }  
9  }  
10  
11  const messenger = new Messenger('...');
```



# КОНСТРУКТОРЫ



# CONSTRUCTOR

`constructor` не является обязательным. Вы можете не создавать его, если он вам не нужен.

# ПОЛЯ

Почему нельзя написать поле выше конструктора, как в других языках (не писать `this.field`)

Данный синтаксис пока поддерживается не во всех браузерах (и, вероятно, будет в стандарте ES2019).

Есть ли инкапсуляция, как в других языках?

На данный момент предполагается, что эта возможность появится в ES2019.

---

# CLASS

Важные моменты:

```
console.log(typeof Messenger); // function  
console.log(Messenger);
```



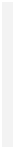
# ОСОБЕННОСТИ

1. Все методы — не перечисляемы:
2. Нельзя использовать без `new`:
3. Нельзя переопределить `prototype`:



# ЗАЧЕМ НУЖНЫ КЛАССЫ?

Зачем нужны классы, если есть функции-конструкторы и прототипы?



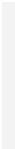
Во-первых, классы позволяют писать более лаконичный код.  
Во-вторых, это современный стиль написания JS-кода.





# ФУНКЦИИ-КОНСТРУКТОРЫ И ЦЕПОЧКИ ПРОТОТИПОВ

Как теперь быть с цепочками прототипов и функциями конструкторов — их больше нет?



Они по-прежнему остались и работают, но скрыты от нас «синтаксическим сахаром»

# ADVANCED

Вопрос:

А что, если я хочу использовать тонкую настройку свойств через `Object.defineProperty`?

Ответ:

Это можно сделать в конструкторе.

# EXTENDS

Позволяет организовать наследование:

```
1 class MultiMessenger extends Messenger { }  
2  
3 const viber = new MultiMessenger('viber');  
4 console.log(viber.name); // viber
```

Все существующие свойства уже «наследуются».



# НАСЛЕДОВАНИЕ

# ДЛЯ ЧЕГО НУЖНО НАСЛЕДОВАНИЕ

Для чего нужно наследование, я ведь могу просто создавать нужные мне классы?

Для переиспользования кода и построения иерархий

Наследование не всегда является хорошим решением, но это вопрос архитектуры

# ДОБАВЛЕНИЕ СВОЙСТВ

```
1 class MultiMessenger extends Messenger {  
2   constructor(name, logo) {  
3     super(name); // <- Messenger.call(this, name): вызов конструктора родителя  
4     this.logo = logo;  
5   }  
6 }
```



# SUPER()

`super()` можно использовать только в конструкторе и только первым вызовом.

# EXTENDS БЕЗ CONSTRUCTOR

На самом деле

```
class MultiMessenger extends Messenger { }
```

Эквивалентно:

```
1 class MultiMessenger extends Messenger {  
2     constructor(...params) {  
3         super(...params);  
4     }  
5 }
```



# ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДА

```
1 class MultiMessenger extends Messenger {  
2     send(recipient, msg) {  
3         // TODO: send message  
4     }  
5 }
```

# ВЫЗОВ РОДИТЕЛЬСКОГО МЕТОДА

```
1 class MultiMessenger extends Messenger {  
2   send(recipient, msg) {  
3     if (<recipient is from our service>) {  
4       super.send(recipient, msg);  
5       return;  
6     }  
7   }  
8 }
```

# SUPER

`super` позволяет получать доступ к свойствам и методам «родителя». Причём не важно, как этот самый родитель был установлен:

```
1  const basic = {  
2    send(msg) {  
3      // TODO: send message  
4    },  
5  };  
6  
7  const viber = {  
8    send(msg) {  
9      super.send(msg);  
10   },  
11  };  
12  
13  Object.setPrototypeOf(viber, basic);  
14  viber.send('hello from viber');
```



# СТАТИЧЕСКИЕ МЕТОДЫ

# СТАТИЧЕСКИЕ МЕТОДЫ

До ES6 статическими методами назывались методы, добавленные в функцию-конструктор:

```
function Image() { ... }  
Image.from = function(...) { ... };
```

В ES6 для этого используется ключевое слово `static`

```
1 class Image {  
2   static from(...) {  
3     ...  
4   }  
5 }
```



## ДРУГИЕ ВОЗМОЖНОСТИ КЛАССОВ

- `get/set`
- передача в функции
- свойства с вычисляемыми именами
- и т.д.



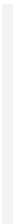
## ES6 STYLE

На сегодняшний день использование классов является предпочтительным.



# ХРАНЕНИЕ КЛАССОВ

Где хранить определения классов — в отдельных файлах или прямо в основном файле приложения?



Это вопрос к организации кода. Поскольку мы используем сборщик (Webpack), то требуем от вас хранения отдельного класса (либо группы связанных классов) в отдельном модуле.





# ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей:

1. Иерархия наследования
2. Классы
3. Конструкторы
4. Наследование
5. Статические методы



Спасибо за внимание!  
Время задавать вопросы 😊

**ЛЕКТОР**

