НЕТОЛОГИЯ

ПРОТОТИПЫ, КОНСТРУКТОРЫ, КЛАССЫ И НАСЛЕДОВАНИЕ

АЛЕКСАНДР ШЛЕЙКО

АЛЕКСАНДР ШЛЕЙКО

Программист в Яндекс

a.shleyko@yandex.ru vk.com/shleiko @dustyo_O

ПЛАН ЗАНЯТИЯ

1. Конструкторы
2. Использование прототипов
3. Привязка контекста
4. Иерархия наследования
5. Классы
6. Наследование

КОНСТРУКТОРЫ

ВОПРОС

Зачем нужны конструкторы?

СОЗДАНИЕ ОБЪЕКТОВ ЧЕРЕЗ ЛИТЕРАЛЫ

Предположим, нужно создать объект товара на странице. Можно использовать литералы:

```
const tshirt = {
  id: 280,
  price: 340,
  name: 'No name t-shirt',
  category: 't-shirt',
};
```

СОЗДАНИЕ ОБЪЕКТОВ ЧЕРЕЗ ЛИТЕРАЛЫ

Через некоторое время у нас будет уже не один товар, а несколько:

```
const iphone = {
  id: 281,
  price: 60000,
  title: 'iPhone 11',
  category: 'mobile',
};
```

Вопрос к аудитории: всё ли в порядке с этим кодом?

ПРОБЛЕМА

Ключевая проблема: мы "случайно" опечатались в названии свойства (в `tshirt` название хранится в свойстве `name`, а в `iphone` в `title`).

Что будет при попытках:

1. Прочитать значение несуществующего свойства?
2. Записать значение в несуществующее свойство?

КОНСТРУКТОР

Решение: выделяем логику создания объекта продукта в функцию.

Конструктор — это функция, инкапсулирующая в себе логику создания объектов определенного типа.

```
function createProduct(id, price, name, category) {
  const product = {
    id,
    price,
    name,
    category,
  };
  return product;
}

const tshirt = createProduct(280, 340, 'No name t-shirt', 't-shirt');
```

ОПЕРАТОР `new`

Оператор `new` выполняет роль синтаксического сахара при создании объектов.

```
// вместо
const product = createProduct(340, 'No name t-shirt', 't-shirt');
// пишем
const tshirt = new Product(280, 340, 'No name t-shirt', 't-shirt');

// сам конструктор:
function Product(id, price, name, category) {
  this.id = id;
  this.price = price;
  this.name = name;
  this.category = category;
}
```

ОСОБЕННОСТИ РАБОТЫ С `new`

Особенности работы функций, вызванных с оператором `new`:

- создаётся новый пустой объект;
- ключевое слово `this` получает ссылку на этот объект;
- функция выполняется;
- возвращается `this`.

`this` возвращается без явного указания!

ВАЖНО

Функции-конструкторы по соглашению принято писать с **Б**ольшой буквы, чтобы показать, что это конструктор.

Но написание функции с большой буквы ничего не даёт. Это по-прежнему функция, которую можно вызывать как с `new`, так и без.

То же самое: любую функцию, написанную с маленькой буквы можно вызывать с `new` и без.

Пример: в стандартной библиотеке есть функция `Number`, которую вы можете вызвать как с `new` (тогда создастся объект), так и без (тогда будет выполнено преобразование аргумента в число).

`this`

Вопросы на засыпку:

1. Куда будет указывать `this`, если конструктор вызвать без `new`?
2. Что будет возвращаться, если в какой-то функции есть `return`, а мы её вызвали с `new`?
3. Если объект создан через `new` и функцию конструктор, можно ли в/из него добавить/удалить свойства?
4. Куда будет указывать `this`, если метод объекта вызвать с `new`?

`this`

Куда будет указывать `this`, если конструктор вызвать без `new`?

Зависит от контекста (об этом чуть позже). В простейшем случае от режима: в strict mode значение будет `undefined`, в не strict - глобальный объект

`this`

Что будет возвращаться, если в какой-то функции есть `return`, а мы её вызвали с `new`?

Это хитрый вопрос. Если вы будете возвращать объект, то вернётся он, если примитив, то вернётся `this`.

`this`

Если объект создан через `new` и функцию конструктор, можно ли в/из него добавить/удалить свойства?

Да.

`this`

Куда будет указывать `this`, если метод объекта вызвать с `new`?

А вот здесь самое интересное: если это метод, объявленный в `class`, о будет сгенерирована ошибка, а если через `prototype`, то - всё как обычно (на пустой объект).

```
'use strict';
function App() { }

App.prototype.demo = function() {
  console.log(this);
}

const app = new App();
const result = new app.demo(); // OK

// реализация на классах

class App {
  demo() {
    console.log(this);
  }
}

const app = new App();
const result = new app.demo(); // TypeError
```

ПРОТОТИПЫ

ВОПРОС

Что такое прототипы и зачем они нужны?

ПРОТОТИПЫ

Прототипы (prototypes) - это механизм, при котором объекты могут переиспользовать методы и свойства других объектов.

Для этого объекты выстраиваются в цепочки (prototype chain).

При этом поиск свойства при чтении осуществляется с учётом цепочки прототипов (т.е. свойство ищется сначала в самом объекте, затем в прототипе объекта и т.д., пока один из прототипов не равен `null`).

ПРОТОТИПЫ

Все экземпляры `Product` нужно создать с единым прототипом.

```
function Product(id, price, name, category) { /*...*/ }
Product.prototype.getShortTitle = function() {
    return `${this.name} - ${this.price}`
};

const tshirt = new Product(...);
const iphone = new Product(...);

console.log(tshirt.getShortTitle === iphone.getShortTitle); // true!
// tshirt.__proto__.getShortTitle === iphone.__proto__.getShortTitle
```

Синтаксис `Product.prototype` позволяет назначать прототип объекта, создаваемого через `new Product(...)`

Вопрос к аудитории: а что происходит при записи?

ПРОТОТИПЫ

При записи цепочка прототипов не играет роли, свойство сразу пишется в тот объект, к которому обращаемся:

```
function Product(id, price, name, category) { /*...*/ }
Product.prototype.getShortTitle = function() {
    return `${this.name} - ${this.price}`
};

const tshirt = new Product(...);
const iphone = new Product(...);

tshirt.getShortTitle = function() {
    return `${this.name} - ${this.price}`
};
// оригинальный метод остался в __proto__.getShortTitle

console.log(tshirt.getShortTitle === iphone.getShortTitle); // false!
// на самом деле: tshirt.getShortTitle === iphone.__proto__.getShortTitle
```

ПОЛИФИЛЛЫ

Не стоит без веской причины переопределять прототипы стандартных объектов или стандартные методы прототипа своими реализациями, например, `toString` (некоторые библиотеки используют эти методы).

Это может привести к неочевидным ошибкам и сложности дальнейшего поддержания кода другими разработчиками.

Исключение: полифиллы.

ПОЛИФИЛЛЫ

В случае, если runtime не поддерживает необходимые методы (например, `closest` для DOM элементов), можно написать свою реализацию, **обязательно проверив отсутствие нативной реализации**.

```
// https://developer.mozilla.org/en-US/docs/Web/API/Element/closest
if (!Element.prototype.matches) {
  Element.prototype.matches = Element.prototype.msMatchesSelector ||
                              Element.prototype.webkitMatchesSelector;
}
if (!Element.prototype.closest) {
  Element.prototype.closest = function(s) {
    var el = this;
    do {
      if (Element.prototype.matches.call(el, s)) return el;
      el = el.parentElement || el.parentNode;
    } while (el !== null && el.nodeType === 1);
    return null;
  };
}
```

ПРОВЕРКА ПРИНАДЛЕЖНОСТИ К КЛАССУ

Для проверки принадлежности экземпляра к определенному типу существет оператор `instanceof`.

Важно: `instanceof` проверяет всю цепочку прототипов.

Пример использования:

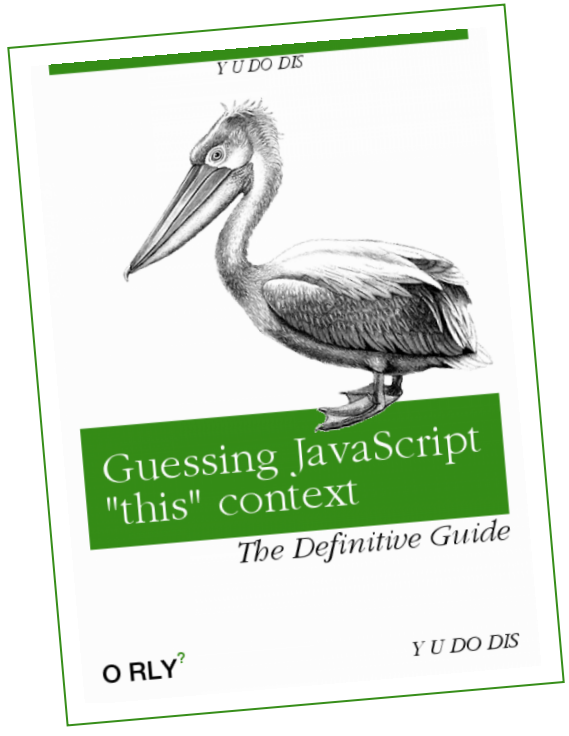
```
const product = new Product(...);

console.log(product instanceof Product); // true
console.log(product instanceof Object); // true
```

ПРИВЯЗКА КОНТЕКСТА

КОНТЕКСТ ВЫПОЛНЕНИЯ ФУНКЦИИ

Context — значение `this`, указывающего на объект, которому «принадлежит» текущий исполняемый код.



«ОДАЛЖИВАНИЕ МЕТОДА»

Частой типичной ошибкой при работе с методами объектов является потеря контекста.

В таком случае `this` внутри функции перестает указывать на объект, к которому он привязан.

```
1 const getShortTitle = product.getShortTitle;
2
3 getShortTitle(); // Uncaught TypeError: Cannot read property 'name' of undefined
```

ПОТЕРЯ КОНТЕКСТА ВЫПОЛНЕНИЯ

В зависимости от способа вызова `getShortTitle`, `this` будет указывать на различные объекты.

```
product.getShortTitle(); // this -> product

const getShortTitle = product.getShortTitle;
getShortTitle();
// this -> глобальный объект
// (window, если не установлено "use strict", помните про модули)
```

Решение: явная привязка контекста исполнения при помощи `bind`, `call` и `apply`.

МЕТОДЫ CALL

Метод прототипа `Function`, вызывающий указанную функцию с привязкой контекста (`this`).

Сигнатура:

```
func.call(context, arg1, arg2, ...)
```

Пример использования:

```
1  const getShortTitle = product.getShortTitle;
2
3  getShortTitle(); // this -> window
4  getShortTitle.call(product); // this -> product
```

МЕТОД APPLY

Выполняет идентичную с `call` фунцию, различается сигнатурой.

Сигнатура `apply`:

```
func.apply(context, [arg1, arg2]);
```

Сигнатура `call`:

```
func.call(context, arg1, arg2, ...)
```

МЕТОД BIND

Метод прототипа `Function`, создающий функцию с привязанным контекстом (`this` code>).

Сигнатура `bind`:

```
func.bind(context, arg1, arg2, ...)
```

Важно: `bind` возвращает новую функцию!

```
1  const getShortTitle = product.getShortTitle;
2
3  const getShortTitleBind = getShortTitle.bind(product); // this -> product
4
5  getShortTitle === getShortTitleBind; // false!
```

ВЫВОДЫ

Что такое `bind`, `apply`, `call` и зачем их использовать?

Методы прототипа `Function` для привязки контекста исполнения к функции. Достаточно часто используются в библиотеках для манипуляции вашим кодом.

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 при чтении сначала ищется в самом объекте, потом в прототипе объекта, потом в прототипе прототипа и т.д.

ЗАДАЧА

Перед нами встала новая 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.

ФУНКЦИЯ КОНСТРУКТОР

Old Style:

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

// prototype - уже объект, поэтому можем просто добавлять свойства
// (но можем и заменить целиком, нарушив предыдущую цепочку)
Messenger.prototype.send = function(recipient, msg) {
  // TODO: send text message
};
```

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

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

function MultiMessenger(name) {
  Messenger.call(this, name); // <-
}
MultiMessenger.prototype = Object.create(Messenger.prototype);
MultiMessenger.prototype.constructor = MultiMessenger;

const viber = new MultiMessenger('Viber');
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:

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

function MultiMessenger(name, logo) {
  Messenger.call(this, name);
  this.logo = logo; // <-
}
MultiMessenger.prototype = Object.create(Messenger.prototype);
MultiMessenger.prototype.constructor = MultiMessenger;

const viber = new MultiMessenger('Viber', 'V');
console.log(viber.name); // Viber
console.log(viber.logo); // V
```

МЕТОДЫ

Посмотрим, что с методами.

НАСЛЕДОВАНИЕ МЕТОДОВ

```
function Messenger() { ... }
Messenger.prototype.send = function(recipient, msg) {
  // TODO: send text message
};

function MultiMessenger() { ... }
MultiMessenger.prototype = Object.create(Messenger.prototype);
MultiMessenger.prototype.constructor = MultiMessenger;

const viber = MultiMessenger();
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`?

```
function Messenger() { ... }
Messenger.prototype.send = function(recipient, msg) {
  // TODO: send text message
};

function MultiMessenger() { ... }
MultiMessenger.prototype = Object.create(Messenger.prototype);
MultiMessenger.prototype.send = function(recipient, msg) { ... };
MultiMessenger.prototype.constructor = MultiMessenger;

const viber = MultiMessenger();
viber.send('...');
```

МЕТОДЫ

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

ВЫЗОВ РОДИТЕЛЬСКОГО МЕТОДА

```
function Messenger() { ... }
Messenger.prototype.send = function(recipient, msg) {
  // TODO: send text message
};

function MultiMessenger() { ... }
MultiMessenger.prototype = Object.create(Messenger.prototype);
MultiMessenger.prototype.send = function(recipient, msg) {
  if (<recipient is from our service>) {
    Messenger.prototype.send.call(this, recipient, msg);
    return;
  }

  // else send via Viber
};
MultiMessenger.prototype.constructor = MultiMessenger;
```

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.

На самом деле, помимо "сахара", `class` имеет другое поведение (что мы видели в начале лекции) нежели `prototype`.

Поэтому говорить, что классы и прототипы - это одно и то же, только записанное по-другому, не совсем верно.

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('...');
```

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

CLASS

Важные моменты:

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

```
function
▼ {constructor: ƒ, send: ƒ}
  ▶ constructor: class Messenger
  ▶ send: ƒ send(recipient, msg)
  ▶ __proto__: Object
```

Особенности

1. Все методы — не перечисляемы:

```
> for (const prop in messenger) {
      console.log(prop);
  }
  name
```

2. Нельзя использовать без `new`:

```
> const bad = Messenger();
✖ ▶Uncaught TypeError: Class constructor Messenger cannot be invoked without 'new'
      at <anonymous>:1:13
```

3. Нельзя переопределить `prototype`:

```
> Messenger.prototype = {};
  ▼{} i
    ▶__proto__: Object
> console.log(Messenger.prototype);
  ▼{constructor: ƒ, send: ƒ} i
    ▶constructor: class Messenger
    ▶send: ƒ send(recipient, msg)
    ▶__proto__: Object
```

ОСОБЕННОСТИ

1. Имеют block-level scope (как `let` с TDZ и всеми вытекающими), в отличие от функций с hoisting (как `var` и всеми вытекающими)
2. Весь код внутри уже в `strict mode`
3. Методы не могут использоваться с `new`

ЗАЧЕМ НУЖНЫ КЛАССЫ?

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

1. Во-первых, классы позволяют писать более лаконичный код.
2. 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.

НАСЛЕДОВАНИЕ

EXTENDS

Позволяет организовать наследование:

```
class MultiMessenger extends Messenger { }

const viber = new MultiMessenger('viber');
console.log(viber.name); // viber
```

Все существующие свойства уже «наследуются».

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

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

SUPER()

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

EXTENDS БЕЗ CONSTRUCTOR

На самом деле

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

Эквивалентно:

```
class MultiMessenger extends Messenger {
  constructor(...params) {
    super(...params);
  }
}
```

ПЕРЕОПРЕДЕЛЕНИЕ МЕТОДА

```
class MultiMessenger extends Messenger {
  send(recipient, msg) {
    // TODO: send message
  }
}
```

ВЫЗОВ РОДИТЕЛЬСКОГО МЕТОДА

```
class MultiMessenger extends Messenger {
  send(recipient, msg) {
    if (<recipient is from our service>) {
      super.send(recipient, msg);
      return;
    }
  }
}
```

SUPER

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

```
const basic = {
  send(msg) {
    // TODO: send message
  },
};

const viber = {
  send(msg) {
    super.send(msg);
  },
};

Object.setPrototypeOf(viber, basic);
viber.send('hello from viber');
```

ДРУГИЕ ВОЗМОЖНОСТИ КЛАССОВ

Мы не будем повторять то, что вы проходили на предыдущих лекциях, но нужно помнить, что классы позволяют сделать ещё больше:

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

Стоит помнить, что 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. Конструкторы
2. Прототипы
3. Привязку контекста
4. Иерархию наследования
5. Классы

НЕТОЛОГИЯ

Задавайте вопросы и напишите отзыв о лекции!

АЛЕКСАНДР ШЛЕЙКО


a.shleyko@yandex.ru


vk.com/shleiko


@dustyo_0