ПРОТОТИПЫ И КОНСТРУКТОРЫ

ЭДГАР НУРУЛЛИН / ФРОНТЕНД РАЗРАБОТЧИК ONETWOTRIP

ЭДГАР НУРУЛЛИН

Фронтенд разработчик OneTwoTrip

EdgarNur

ПЛАН ЗАНЯТИЯ

1. Конструкторы
2. Использование прототипов
3. Привязка контекста
4. Документирование кода

КОНСТРУКТОРЫ

СОЗДАНИЕ ОБЪЕКТОВ ЧЕРЕЗ ЛИТЕРАЛЫ

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

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

if (product.category === 't-shirt') {
  product.sizeTable = SIZE_TABLES.T_SHIRTS;
}
```

СОЗДАНИЕ ОБЪЕКТОВ ЧЕРЕЗ ЛИТЕРАЛЫ

Через некоторое время нужно будет написать корзину. Понадобится создавать объекты товаров:

```
cart.products.map(rawProduct => {
  const product = {
    price: rawProduct.price,
    name: rawProduct.originalName,
    category: rawProduct.categoryName,
  };

  if (rawProduct.categoryName === 't-shirt') {
    product.sizeTable = SIZE_TABLES.T_SHIRTS;
  }
});
```

НАРУШЕНИЕ ПРИНЦИПА DRY

Мы видим, что логика создания объекта дублируется, нарушается принцип DRY — Don't repeat youself («Не повторяйся»).

В дальнейшем изменение структуры или названий полей будет осложняться.

Решение: выделяем логику создания объекта продукта в функцию.

КОНСТРУКТОР

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

```
function createProduct(price, name, category) {
  const product = {
    price,
    name,
    category,
  };
  if (category === 't-shirt') {
    product.sizeTable = SIZE_TABLES.T_SHIRTS;
  }
  return product;
}

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

ОПЕРАТОР `new`

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

Вызов конструктора...

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

...визуально упрощается:

```
const product = new Product(340, 'No name t-shirt', 't-shirt');
```

ОПЕРАТОР `new`

Упрощается и конструктор:

```
function Product(price, name, category) {
  this.price = price;
  this.name = name;
  this.category = category;

  if (category === 't-shirt') {
    this.sizeTable = SIZE_TABLES.T_SHIRTS;
  }
}
```

ОСОБЕННОСТИ РАБОТЫ КОНСТРУКТОРОВ С `new`

Особенности работы функций, вызванных через оператор `new`:

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

`this` возвращается без явного указания!

ВЫВОДЫ

Зачем нужны конструкторы, если я могу использовать объектные литералы?

Чтобы инкапсулировать логику создания объекта определенного класса.

ВЫВОДЫ

Зачем нужны конструкторы, если я потом в любое время могу добавить/удалить свойство?

Такой код будет проще читать другому разработчику, вся логика будет собрана в одном месте.

ИСПОЛЬЗОВАНИЕ ПРОТОТИПОВ

ДОБАВЛЕНИЕ МЕТОДОВ

Через некоторое время при написании корзины понадобилось добавить объектам `Product` метод.

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

ИЗЛИШНИЕ РАСХОДЫ ПАМЯТИ

Недостаток рассмотренного подхода состоит в том, что если будет множество экземпляров `Product`, это приведет к дублированию в памяти метода `getShortTitle`.

```
1  const product1 = new Product(...);
2  const product2 = new Product(...);
3
4  product1.getShortTitle === product2.getShortTitle; // false!
```

Решение: использовать прототипы.

ВСПОМНИМ О ПРОТОТИПАХ

Все экземпляры `Product` нужно создать с единым прототипом.

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

const product1 = new Product(...);
const product2 = new Product(...);

product1.getShortTitle === product2.getShortTitle; // true!
```

PROTOTYPE

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

MONKEY PRACTICE ПО PROTOTYPE

Нельзя переопределять в прототипе стандартные методы своими реализациями, например, `toString`.

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

Исключение: полифиллы.

ПОЛИФИЛЛЫ

В случае, если браузер устаревший и не поддерживает необходимые методы (например, в IE8 не поддерживается getter `firstElementChild` для DOM элементов), вы можете написать свою реализацию, **обязательно проверив отсутствие нативной реализации**.

```
if (document.documentElement.firstElementChild === undefined) {
  Object.defineProperty(Element.prototype, 'firstElementChild', {
    get: function() {
      // код полифилла
    }
  });
}
```

ПРОВЕРКА ПРИНАДЛЕЖНОСТИ К КЛАССУ

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

Важно: `instanceof` проверяет всю цепочку прототипов.

Пример использования:

```
1 const product = new Product(...);
2
3 product instanceof Product; // true
```

ПРОВЕРКА НАЛИЧИЯ СВОЙСТВА У ОБЪЕКТА

Для проверки наличия свойства у объекта, а не у его прототипа, предусмотрен метод протипа объекта `hasOwnProperty`.

Пример использования:

```
product.hasOwnProperty(price); // true
product.hasOwnProperty(getShortTitle); // false
```

ВЫВОДЫ

Зачем мне прототипы, если я могу прямо в конструкторе добавлять в объект функции (методы)?

Для экономии памяти, при использовании прототипа у всех представителей класса будет один и тот же экземпляр метода.

ВЫВОДЫ

Что будет, если я заменю функцию в прототипе одного из «встроенных» объектов (объектов стандартной библиотеки)?

Возможно появится не предусмотренное поведение кода, в местах, где вы этого не ожидаете.

ВЫВОДЫ

Когда это стоит делать?

Только для реализации полифиллов.

ВЫВОДЫ

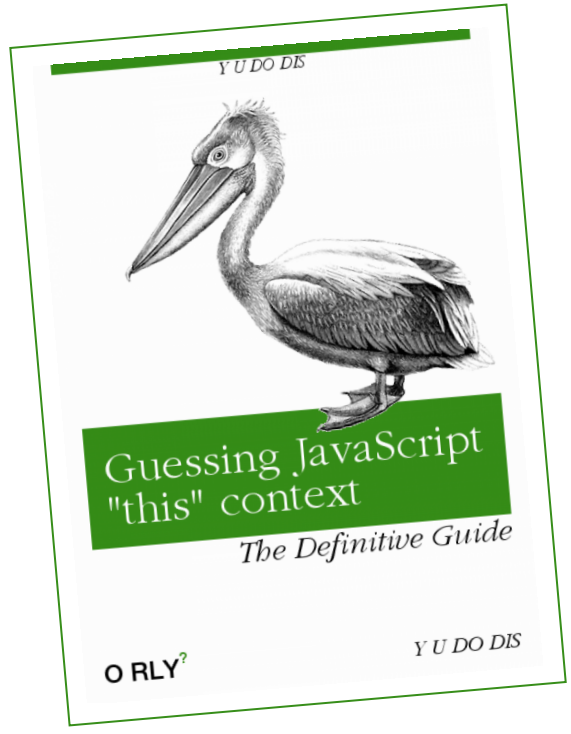
Когда нужно писать свои методы в прототипе?

Если метод будет один и тот же для всех экземпляров класса.

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"
```

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

Сигнатура `apply` :

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

Важно: `bind` вызывает новую функцию!

```
1  const getShortTitle = product.getShortTitle;
2
3  const getShortTitle2 = getShortTitle.bind(product); // this -> product
4
5  getShortTitle === getShortTitle2; // false!
```

ПРОСТАЯ РЕАЛИЗАЦИЯ BIND

Для понимания работы можно рассмотреть простую реализацию `bind` через `apply`.

```
function bind(func, context) {
  return function() {
    return func.apply(context, arguments);
  };
}
```

ВЫВОДЫ

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

Методы прототипа `Function` для привязки контекста исполнения к функции.

ДОКУМЕНТИРОВАНИЕ КОДА

ДОКУМЕНТИРОВАНИЕ КОДА

Когда вы работаете в команде, ключевое значение приобретает **knowledge sharing** — распространение знаний между участниками команды.

То же самое происходит, когда вы хотите использовать сторонний фреймворк или библиотеку — наличие информации об использовании играет ключевую роль.

ПИШИ КОД ТАК, ЧТОБЫ НЕ НУЖНА БЫЛА ДОКУМЕНТАЦИЯ

Это верно, но в большинстве случаев не отменяет необходимости наличия документации. Давайте рассмотрим, какие инструменты есть в мире JS и научимся ими пользоваться.

DOC COMMENTS

Большинство систем документирования работают схожим образом и предлагают нам блочные комментарии, часть символов в которых интерпретируется специальным образом:

```
/**
 * Конструктор пользователя
 *
 * @param name имя пользователя
 * @param type тип пользователя
 */
function User(name, type) {
  this.name = name;
  this.type = type;
}
```

DOC COMMENTS

Затем эти комментарии обрабатываются инструментом для извлечения документации.

ИНСТРУМЕНТЫ

- JSDoc;
- ESDoc;
- DocumentJS;
- и другие.

Если внимательно приглядеться, то большинство «тегов» повторяются во всех системах: `@param`, `@return`, `@throws` и т.д.

JSDOC

Установим и настроим JSDoc:

```
$ npm install --save-dev jsdoc
```

JSDOC

Создадим конфигурационный файл (`jsdoc.conf.json`):

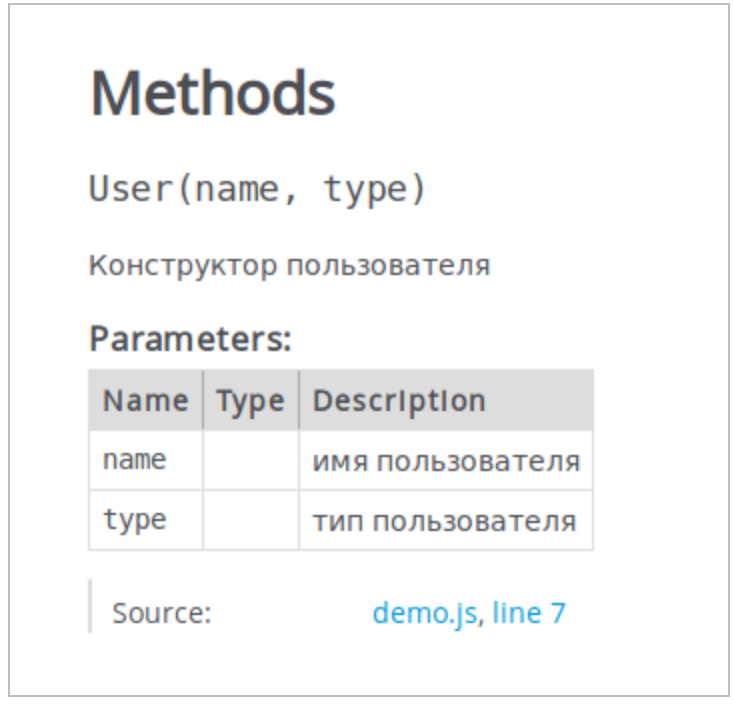
```
{
  "source": {
    "include": ["src/js"]
  },
  "opts": {
    "encoding": "utf8",
    "destination": "./docs/",
    "recurse": true
  }
}
```

Не забудьте добавить каталог `docs` в `.gitignore` и `.eslintignore`.

JSDOC - ГЕНЕРАЦИЯ

Добавим в `package.json` скрипт:

```
"scripts": {
  ...
  "doc": "jsdoc -c jsdoc.conf.json"
},
```



Methods

`User(name, type)`

Конструктор пользователя

Parameters:

Name	Type	Description
name		имя пользователя
type		тип пользователя

Source: demo.js, line 7

ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей:

1. **Конструкторы**

Для инкапсуляции логики создания объекта определённого класса.

2. **Прототипы**

Для того, чтобы у всех представителей класса был один и тот же экземпляр метода.

3. **Привязку контекста**

Чтобы `this` внутри функции не переставал указывать на объект, к которому он привязан.

4. **Документирование кода**

Чтобы облегчить работу с кодом.

Спасибо за внимание! Время задавать вопросы!

ЭДГАР НУРУЛЛИН

EdgarNur