

СВОЙСТВА ОБЪЕКТА, FOR-IN, ОБЁРТКИ ДЛЯ ПРИМИТИВОВ



МИХАИЛ ЛАРЧЕНКО / TECH LEAD



МИХАИЛ ЛАРЧЕНКО

Tech Lead



larchanka@me.com

ПЛАН ЗАНЯТИЯ

- Объекты
- Свойства объекта
- Прототипы и цепочки прототипов
- Object
- Перебор свойств
- get/set
- Обёртки для примитивов

ОБЪЕКТЫ

ОБЪЕКТЫ

В JS объекты представляют из себя набор свойств (пар ключ–значение).

Вопрос к аудитории:

Зачем нам нужны объекты?

СВОЙСТВА ОБЪЕКТА

СВОЙСТВА ОБЪЕКТА

Вспомним варианты доступа к свойствам объекта:

```
1 | const user = {  
2 |   name: 'Nemo',  
3 |   balance: 10000,  
4 | };  
5 | // Вариант 1: 'dot notation'  
6 | console.log(user.name);  
7 | // Вариант 2: 'bracket notation'  
8 | console.log(user['name']);
```

Вопрос к аудитории:

Когда и какой способ предпочтительнее?

ИЗВЛЕЧЕНИЕ СВОЙСТВ

ES6 предоставляет нам удобный способ извлечения свойств с помощью **Object Destructing**:

```
const {name, balance} = user;
```

ДОБАВЛЕНИЕ И УДАЛЕНИЕ СВОЙСТВ

В JS мы в любой момент можем как добавить объекту новое свойство, так и удалить его:

```
1 user.address = '...';
2 user['address'] = '...';
3
4 delete user.address;
```

ДОСТУП К НЕСУЩЕСТВУЮЩИМ СВОЙСТВАМ

Если мы удалили свойство у объекта (или его просто никогда в объекте не было), то попытка доступа закончится тем, что мы получим `undefined`:

```
console.log(user.address);  
// undefined
```

ПРОВЕРКА НА `undefined`

Вопрос к аудитории:

Чем плох следующий код?

```
1 | if (user.address === undefined) {  
2 |     // No such property  
3 | }
```

ПРОВЕРКА НА `undefined`

На самом деле, свойство в объекте может и быть, а его значение может быть равным `undefined`. Тогда проверка и последующая логика будут некорректной.

OBJECT DESTRUCTURING: DEFAULT VALUES

При **Object Destructuring** мы можем назначать переменным default-значения, если таких полей в объекте нет:

```
const {name, balance, address = 'Не указан'} = user;
```

NESTED OBJECT DESTRUCTURING

А что если объект включает в себя свойство, представляющее из себя объект. И нам нужно извлечь свойства из этого объекта? Поможет ли **Object Destructuring**?

```
1 user.manager = {  
2     name: 'Светлана',  
3     ...  
4 };  
5  
6 const {manager: {name}} = user;
```

Но при этом имя менеджера сохранится в переменную `name` (а такая уже создана).

ПЕРЕИМЕНОВАНИЕ ПРИ ОБЪЕКТ DESTRUCTURING

```
const {manager: {name: managerName}} = user;
```

Теперь имя менеджера сохранится в переменную `managerName`;

Удалим свойство `manager`, чтобы оно нам в дальнейшем не мешало:

```
delete user.manager;
```

REST

В ES2018 появилась возможность использовать конструкцию `...rest` при `Object destruction`:

```
const {name, ...rest} = user;
```

В `rest` будет:

```
1 {  
2   "balance": 10000  
3 }
```

REST

Это даёт замечательные возможности по созданию **Shallow Copy** для объектов:

```
const copy = { ...user};
```

А также для объединения нескольких объектов в один:

```
const merged = { ...first, ...second};
```

ЗАДАЧА

Представим, что мы реализуем CRM-систему, где объекту можно добавлять произвольные поля, например:

1. Ответственный
2. Приоритет
3. Категория
4. и т.д.

Т.е. у одних объектов такие свойства могут быть, а других – нет.
Как найти все объекты, у которых есть определённое свойство?

IN

Оператор `in` позволяет проверить наличие свойства в объекте*:

```
1 | console.log('name' in user); // true
2 | console.log('address' in user); // false
3 | console.log('toString' in user); // true!
```

С первыми двумя примерами всё понятно, но почему в последнем `true`?

ПРОТОТИПЫ И ЦЕПОЧКИ ПРОТОТИПОВ

ПРОТОТИПЫ

JS – объектно-ориентированный язык, основанный на прототипах

Т.е. у каждого объекта есть специальное свойство `__proto__`, в котором может находиться другой объект. И когда мы пытаемся обратиться к определённому свойству нашего объекта, то JS сначала ищет это свойство в нашем объекте, потом в прототипе, потом в прототипе прототипа и т.д.

ПРОТОТИПЫ

```
console.log(user.__proto__);
```

```
> console.log(user.__proto__)
▼ {constructor: f, __defineGetter__: f, __defineSetter__: f, hasOwnProperty: f, __lookupGetter__: f, ...} ⓘ
  ► constructor: f Object()
  ► hasOwnProperty: f hasOwnProperty()
  ► isPrototypeOf: f isPrototypeOf()
  ► propertyIsEnumerable: f propertyIsEnumerable()
  ► toLocaleString: f toLocaleString()
  ► toString: f toString()
  ► valueOf: f valueOf()
  ► __defineGetter__: f __defineGetter__()
  ► __defineSetter__: f __defineSetter__()
  ► __lookupGetter__: f __lookupGetter__()
  ► __lookupSetter__: f __lookupSetter__()
  ► get __proto__: f __proto__()
  ► set __proto__: f __proto__()
```

ЛИТЕРАЛЬНАЯ ФОРМА И ПРОТОТИПЫ

Если вы создаёте объект с помощью литерала, то его прототипом автоматически назначается объект типа `Object`, в котором и определено свойство `toString`.

Если быть точнее, то `Object.prototype`.

ПРОТОТИПЫ

```
console.log(user.__proto__.__proto__);
```

```
> console.log(user.__proto__.__proto__)  
null
```

OBJECT

ОБЪЕКТ

JS содержит встроенный объект [Object](#), который содержит ряд полезных методов для работы с объектами.

В частности, он содержит статический метод `setPrototypeOf`, который позволяет заменить прототип объекта.

```
1 const entry = {  
2   id: 999,  
3 };  
4  
5 Object.setPrototypeOf(user, entry);  
6  
7 console.log(user.id);
```

Не используйте метод `setPrototypeOf` в production-коде. Мы его используем только для демонстрации концепций языка

ЦЕПОЧКА ПРОТОТИПОВ

```
> console.log(user)
▼ {name: "Nemo", balance: 10000} ⓘ
  balance: 10000
  name: "Nemo"
  ▼ __proto__:
    id: 999
    ▼ __proto__:
      ► constructor: f Object()
      ► hasOwnProperty: f hasOwnProperty()
      ► isPrototypeOf: f isPrototypeOf()
      ► propertyIsEnumerable: f propertyIsEnumerable()
      ► toLocaleString: f toLocaleString()
      ► toString: f toString()
      ► valueOf: f valueOf()
      ► __defineGetter__: f __defineGetter__()
      ► __defineSetter__: f __defineSetter__()
      ► __lookupGetter__: f __lookupGetter__()
      ► __lookupSetter__: f __lookupSetter__()
      ► get __proto__: f __proto__()
      ► set __proto__: f __proto__()
```

ЗАДАЧА

Хорошо, мы разобрались с тем, как проверить, есть свойство или нет (правда осталась проблема с прототипами), но что, если нам нужно вывести все свойства?

Например, мы хотим отобразить все свойства объекта в карточке (в виде таблички).

ПЕРЕБОР СВОЙСТВ

ПЕРЕБОР СВОЙСТВ

Итак, мы посмотрели на оператор `in`, который позволяет проверять наличие свойства в объекте (включая цепочку прототипа), давайте посмотрим на `for...in`:

```
1 | for (const prop in user) {  
2 |   console.log(prop);  
3 | }
```

```
> for (const prop in user) {  
    console.log(prop);  
}  
name  
balance  
id
```

HASOWNPROPERTIES

Прототип `Object.prototype` дарит каждому объекту метод `hasOwnProperty`, который позволяет определить, принадлежит ли свойство нашему объекту или берётся из цепочки прототипов:

```
1 for (const prop in user) {  
2     if (user.hasOwnProperty(prop)) {  
3         console.log(prop);  
4     }  
5 }
```

```
> for (const prop in user) {  
    if (user.hasOwnProperty(prop)) {  
        console.log(prop);  
    }  
}  
name  
balance
```

НЕЛЬЗЯ ЛИ ПОПРОЩЕ?

Можно, есть статический метод `Object.keys`, который массив собственных перечисляемых свойств (не включая цепочку прототипов).

OBJECT.DEFINEPROPERTY & OBJECT.DEFINEPROPERTIES

При создании свойства в объекте, мы можем определить ряд характеристик (дескриптор), которые определяют поведение этого свойства. Вот, что по этому поводу говорит [MDN](#):

- `configurable` – свойство может быть удалено из содержащего его объекта
- `enumerable` – свойство можно увидеть через перечисление свойств
- `value` – значение, ассоциированное со свойством
- `writable` – значение, ассоциированное со свойством, может быть изменено с помощью оператора `=`
- `get` – функция, используемая как `getter` свойства
- `set` – функция, используемая как `setter` свойства

OBJECT.GETOWNPROPERTYDESCRIPTOR

Посмотрим на дескриптор собственных свойств объекта `user`:

```
console.log(Object.getOwnPropertyDescriptor(user, 'name'));
```

```
> console.log(Object.getOwnPropertyDescriptor(user, 'name'));
```

```
▼ {value: "Nemo", writable: true, enumerable: true, configurable: true} ⓘ  
  configurable: true  
  enumerable: true  
  value: "Nemo"  
  writable: true  
▶ __proto__: Object
```

OBJECT.GETOWNPROPERTYDESCRIPTOR

Вот и ответ на вопрос, почему в `for..in` мы не видели некоторых свойств:

```
> console.log(Object.getOwnPropertyDescriptor(user.__proto__.__proto__, 'toString'));  
▼ {value: f, writable: true, enumerable: false, configurable: true} ⓘ  
  configurable: true  
  enumerable: false  
▶ value: f toString()  
  writable: true  
▶ __proto__: Object
```

КАК ПЕРЕБРАТЬ ВСЕ СВОЙСТВА ОБЪЕКТА?

Если вам нужны все перечисляемые, включая цепочку прототипов,
то через `for..in`.

Если вам нужны все перечисляемые собственные,
то `for..in` + `hasOwnProperty` (либо `Object.keys`)

TOSTRING

Что происходит, когда мы пытаемся использовать объект в «строковом контексте»?

```
console.log(`Current user: ${user}`);
// Current user: [object Object]
```

На самом деле вызывается метод `toString`, который определён в цепочке прототипов.

Что будет, если мы напишем свой метод `toString`? Тогда по правилам JS сначала будет искать это свойство в нашем объекте и только если не найдёт – пойдёт искать по цепочке.

TOSTRING

```
1 | user.toString = function() {  
2 |   return `User ${this.name}`;  
3 | };
```

```
> console.log(`Current user: ${user}`);  
Current user: User{Nemo}
```

СТРЕЛОЧНЫЕ ФУНКЦИИ

Вопрос к аудитории:

Почему это не сработает?

```
1 | user.toString = () => {
2 |   return `User ${this.name}`;
3 | };
```

КАК ПРАВИЛЬНО ОБЪЯВЛЯТЬ МЕТОДЫ?

Попробуем создать новый объект, в котором сразу в лiteralной форме прописать метод.

Вариант 1:

```
1 const good = {  
2     code: '45007',  
3     name: 'Стильный чехол',  
4     description: '...',  
5     price: 1500,  
6     toString: function() {  
7         return `[$this.code] ${this.name} за ${this.price} руб.`  
8     },  
9 };
```

КАК ПРАВИЛЬНО ОБЪЯВЛЯТЬ МЕТОДЫ?

ES2015 (либо транспайлеры) позволяют нам использовать сокращённый синтаксис.

Вариант 2:

```
1 const good = {  
2     code: '45007',  
3     name: 'Стильный чехол',  
4     description: '...',  
5     price: 1500,  
6     toString() { // ES2015  
7         return `[${this.code}] ${this.name} за ${this.price} руб.`  
8     },  
9 };
```

Старайтесь использовать более новый синтаксис (при наличии возможности).

ЗАДАЧА

Возникает необходимость сравнения двух объектов (например, при поиске или сортировке). Варианты решения:

1. Сравнение свойств
2. `valueOf`

Со сравнением свойств всё понятно, рассмотрим `valueOf`.

VALUEOF

Метод прототипа, вызывающийся при преобразовании объекта к примитивному типу (не к строковому контексту).

Например:

```
1 const project1 = { ... };
2 const project2 = { ... };
3
4 if (project1 > project2) {
5     // TODO:
6 }
```

Переопределение `valueOf` позволяет нам задать «собственные правила сравнения».

КАК СРАВНИВАТЬ ОБЪЕКТЫ НА РАВЕНСТВО?

Только через сравнение полей.

Если хотите сравнивать в контексте приведения к примитивным типам
(либо приводить к ним), то переопределяйте `valueOf`.

ДРУГИЕ ПОЛЕЗНЫЕ МЕТОДЫ ОБЪЕКТ

- Object.create(proto) – позволяет создать объект, используя объект (proto) в качестве прототипа
- Object.getPrototypeOf(obj) – получение прототипа объекта

ЗАДАЧА

Нужно разработать приложение, которое позволяет задавать гео-метки, причём делать это не только на карте, а указывая пару координат в виде строки в формате: '55.7887400, 49.1221400', где первое – это широта, второе – долгота.

Как это реализовать?

СВОЙСТВО

```
1 const mark = {  
2   coords: ...  
3 };  
4  
5 mark.coords = '55.7887400, 49.1221400';
```

Но как тогда получать в удобном виде отдельно широту, отдельно долготу?
И как построить логику обработки ошибок?

МЕТОДЫ

```
1 const mark = {  
2   latitude: ...,  
3   longitude: ....,  
4   setCoords(value) {  
5     ...  
6   },  
7   getCoords() {  
8     ...  
9   },  
10};  
11  
12 mark.setCoords('55.7887400, 49.1221400');
```

GET/SET

GET/SET

ES6 появилась возможность сделать подобные методы (их называют *access property*) в более удобном виде (скрыв факт того, что будет вызываться метод).

```
1 const mark = {  
2     latitude: ....,  
3     longitude: ....,  
4     set coords(value) {  
5         ...  
6     },  
7     get coords() {  
8         ...  
9     },  
10};  
11  
12 mark.coords = '55.7887400, 49.1221400'; // вызов setter'a  
13 console.log(mark.coords); // вызов getter'a
```

ОБЁРТКИ ДЛЯ ПРИМИТИВОВ

ОБЁРТКИ ДЛЯ ПРИМИТИВОВ

В JS все типы данных делятся на два больших класса:

- примитивы
- объекты

Например, строка – это примитив. Но при этом возможно следующее:

```
console.log('Hello world'.length);
```

Оператор `.` – обращение к свойству объекта, но строка – это примитив, а не объект.

КАК ЭТО ПРОИСХОДИТ?

JS при использовании примитива с операторами, работающими с объектами, производит следующую операцию: неявно заворачивает примитив в соответствующий объектный тип. Например, для строк – это [String](#).

Для чисел и boolean это будет `Number` и `Boolean`, соответственно.

ЗАЧЕМ НУЖНЫ ОБЁРТКИ?

Обёртки содержат полезные свойства для конкретного типа данных:

- для строк – методы поиска, замены, конвертация регистра;
- для чисел – методы преобразования (различные системы счисления), константы, хранящие max и min-допустимые значения.

ЧЕМ ОТЛИЧАЮТСЯ ОБЁРТКИ ОТ ПРИМИТИВОВ?

Оператор `typeof` вернёт разные значения для примитива и объекта.

Оператор `====` при сравнении объекта (даже если внутри то же значение) с примитивом вернёт `false`.

ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей:

1. Объекты
2. Свойства объекта
3. Прототипы и цепочки прототипов
4. Object
5. Перебор свойств
6. get/set
7. Обёртки для примитивов



Спасибо за внимание!

Время задавать вопросы 😊

МИХАИЛ ЛАРЧЕНКО



larchanka@me.com