

OBJECT, REFLECTION, PROXY



ДЕНИС ЕЖКОВ



ДЕНИС ЕЖКОВ

Frontend-разработчик в «Ростелеком IT»





ПЛАН ЗАНЯТИЯ

- Объекты
- Свойства объекта
- Прототипы и цепочки прототипов
- Object
- Перебор свойств
- Proxy & Reflect



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 Destructuring**:

```
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 OBJECT 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** (поверхностная копия) 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
```

Конечно, правильнее использовать специализированный метод `Object.getPrototypeOf(obj)`, чем обращаться напрямую к полю с двумя подчёркиваниями.



OBJECT

OBJECT

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__()
```



OBJECT

Правильнее было бы использовать специальный метод `Object.create(proto)`, который позволяет создавать объект с нужным прототипом.

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 возвращает массив **имён** собственных перечисляемых свойств (не включая цепочку прототипов): `['balance', 'name']`

А метод `Object.values`, возвращает массив **значений** собственных перечисляемых свойств (не включая цепочку прототипов): `[10000, 'Nemo']`

Метод `Object.entries`, возвращает массив собственных перечисляемых свойств (не включая цепочку прототипов) уже в формате **пар ключ-значение**: `[['balance', 10000], ['name', 'Nemo']]`

OBJECT.DEFINEPROPERTY

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
```

OBJECT.DEFINEPROPERTY

Вызов `Object.defineProperty` с передачей имени уже существующего свойства приведёт к его переконфигурации (т.е. новое свойство создано не будет, будет изменено существующее).

Причём если вы передадите объект, который содержит только ряд полей (например `{configurable: false}`), то остальные значения дескриптора останутся по умолчанию.

Q & A

Q: т.е. на самом деле мы всегда может добавить в объект новое свойство, а вот удалить только при `configurable = false`?

A: не совсем.

FREEZE, SEAL, PREVENTEXTENSION

Есть ряд методов в `Object`, которые позволяют пойти дальше отдельных свойств и наложить ограничения на сам объект:

- `Object.freeze(user)` - "замораживает объект": нельзя модифицировать свойства (включая добавление и удаление), менять дескрипторы и изменять прототип
- `Object.seal(user)` - "пломбирует объект": то же, что и `freeze`, но можно изменять значение существующих свойств (если они `writable`)
- `Object.preventExtension(user)` - то же, что и `seal`, но можно удалять существующие свойства

Важно: попытки выполнения недопустимых действий в режиме `use strict` будут вызывать ошибки.

Для проверки есть соответствующие методы с префиксом `is`: `isFrozen` и т.д.

ИТОГО

Q: как перебрать все свойства объекта?

A: зависит от ваших целей:

- если вам нужны все перечисляемые, включая цепочку прототипов, то через `for..in`.
- если вам нужны все перечисляемые собственные, то `for..in` + `hasOwnProperty` (либо `Object.entries` и т.д.)

Q: можно ли в объект добавить/удалить свойство?

A: зависит от того, как было объявлено свойство и вызывался ли на объекте `Object.freeze` или `Object.seal`

ПОЧЕМУ ЭТО ВАЖНО

Во-первых, это позволяет вам понять, как устроен сам язык и как работают библиотеки (например, библиотека [Immutable.js](https://immutable-js.github.io/immutable-js/) активно использует дескрипторы свойств).

Во-вторых, это очень любят спрашивать на собеседованиях, проверяя то, насколько вы "погружались" в сам язык.

Поэтому мы рассмотрим ещё ряд вещей подобного рода.

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 работает?

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

КАК ПРАВИЛЬНО ОБЪЯВЛЯТЬ МЕТОДЫ?

Попробуем создать новый объект, в котором сразу в литеральной 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`.

```
Object.is
```



OBJECT.IS

Вы часто будете наталкиваться на этот метод в описании работы библиотек. Например (выдержка из документации Jest):

toBe uses Object.is to test exact equality. If you want to check the value of an object, use toEqual instead.

OBJECT.IS

Поэтому нужно знать, как он работает: возвращается `Object.is(a, b) === true`, в случаях:

1. a и b равно undefined
2. a и b равно null
3. a и b равно true или a и b равно false
4. a и b строки с одинаковым содержимым и длиной
5. a и b указывают на один объект
6. a и b число и равны +0, a и b число и равны -0
7. a и b число и равны NaN
8. a и b число и равны одному и тому же числу (но не +/-0 или NaN)

Это важно: в JS +0 и -0 - это разные числа. `===` это игнорирует, а `Object.is` - нет.



PROXY & REFLECT

PROXY

Proxy - это один из широко распространённых методов в разработке, который позволяет "подкладывать" объект, перехватывающий вызовы к оригинальному объекту.

В зависимости от целей **Proxy** может, например, только анализировать вызовы (логгировать их), либо изменять их поведение.

Общая схема выглядит следующим образом:

вызывающий код -> proxy -> оригинальный объект (target)

Т.е. proxy может перехватывать большинство вызовов и либо перенаправлять их оригинальному объекту, либо обрабатывать на своём уровне.

PROXY

Какие вызовы может перехватывать Proxy:

- `get` - чтение свойства
- `set` - запись свойства
- `has` - оператор `in`
- `deleteProperty` - оператор `delete`
- `getPrototypeOf` - вызов `Object.getPrototypeOf`
- `setPrototypeOf` - вызов `Object.setPrototypeOf`
- `isExtensible` - вызов `Object.isExtensible`
- `preventExtension` - вызов `Object.preventExtension`
- `getOwnPropertyDescriptor` - вызов `Object.getOwnPropertyDescriptor`
- `defineProperty` - вызов `Object.defineProperty`
- `ownKeys` - вызов `Object.keys`, `Object.getOwnPropertyNames`, `Object.getOwnPropertySymbols`
- `apply` - вызов функции
- `construct` - вызов функции с `new`

PROXY

Пример:

```
> const user = {name: 'Nemo', balance: 10000};
< undefined

> const proxy = new Proxy(user, {
  get(target, key, receiver) {
    console.log(target); console.log(key);
    Reflect.get(target, key, receiver);
  },
  set(target, key, value, receiver) {
    console.log(target); console.log(key, value);
    return Reflect.set(target, key, value, receiver);
  },
});
< undefined

> proxy.balance = 50000;
  ► {name: "Nemo", balance: 10000}
  balance 50000
< 50000
```

Где:

- `target` - оригинальный объект `user`
- `receiver` - контекст вызова (`this` - сам прокси, либо объект, у которого прокси в цепочке прототипов)

REFLECTION

Рефлексия - это возможность анализировать приложение (например, объекты) во время исполнения (т.е. не в момент написания, а именно в момент исполнения).

В JS рефлексия была встроена изначально, поскольку у нас есть возможность анализировать доступные свойства, устанавливать им значения через название свойства, хранящегося в переменной.

Изначально все методы (речь именно про методы, а не операторы типа `in`) для рефлексии "копились" в `Object`, но их решили немного упорядочить и скомпоновали в объект `Reflect` (при этом изменив часть поведения: там, где вызовы методов `Object` генерируют ошибки, вызовы `Reflect` возвращают `true` или `false`).

REFLECT

`Reflect` - это просто удобный объект со статическими методами, который позволяет "вызывать поведение по умолчанию" вместо "ручного вызова" (например, вместо `target[key] = value` написать `Reflect.set(target, key, value)` и т.д.)

Все методы `Proxy`, которые мы рассмотрели, имеют соответствующие статические методы в объекте `Reflect`, что позволяет достаточно единообразно описывать проху:

```
1  const proxy = new Proxy(user, {
2    get(target, key, receiver) {
3      // т.е. нам достаточно повторить сигнатуру выше
4      // но уже на объекте Reflect
5      Reflect.get(target, key, receiver);
6    },
7    set(target, key, value, receiver) {
8      // т.е. нам достаточно повторить сигнатуру выше
9      // но уже на объекте Reflect
10     return Reflect.set(target, key, value, receiver);
11   },
12 });
```

PROXY

Важно внимательно читать [сигнатуру методов](#) `Proxy`. Например, в случае успешной установки значения `set` мы должны возвращать `true`.

Стоит отметить, что `Proxy` не всегда может перехватить вызовы всех методов (например, для встроенных объектов `Array`), но это выходит за рамки нашей лекции.

ЗАЧЕМ ЭТО НУЖНО (PROXY + REFLECT)

Нужно это в первую очередь, для двух целей:

1. Отладки кода - чтобы посмотреть, какие функции (включая `Object.*`) библиотеки вызывают на вашем объекте, и что может пойти не так
2. Написания библиотек - возможность перехватывать обращения в методам позволяет встраивать собственную логику (так называемое аспектно-ориентированное программирование), внедряя логгирование, контроль доступа и т.д.



ИТОГИ

Сегодня мы с вами рассмотрели достаточно много важных вещей:

1. Объекты
2. Свойства объекта
3. Object
4. Перебор свойств
5. Proxy и Reflect



Задавайте вопросы и напишите отзыв о лекции!

ДЕНИС ЕЖКОВ

 [aka_SKIff](https://t.me/aka_SKIff)

 facebook.com/ezhkov

 [@ezhkov](https://t.me/@ezhkov)