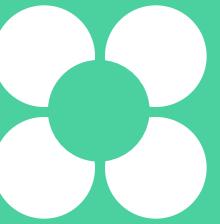
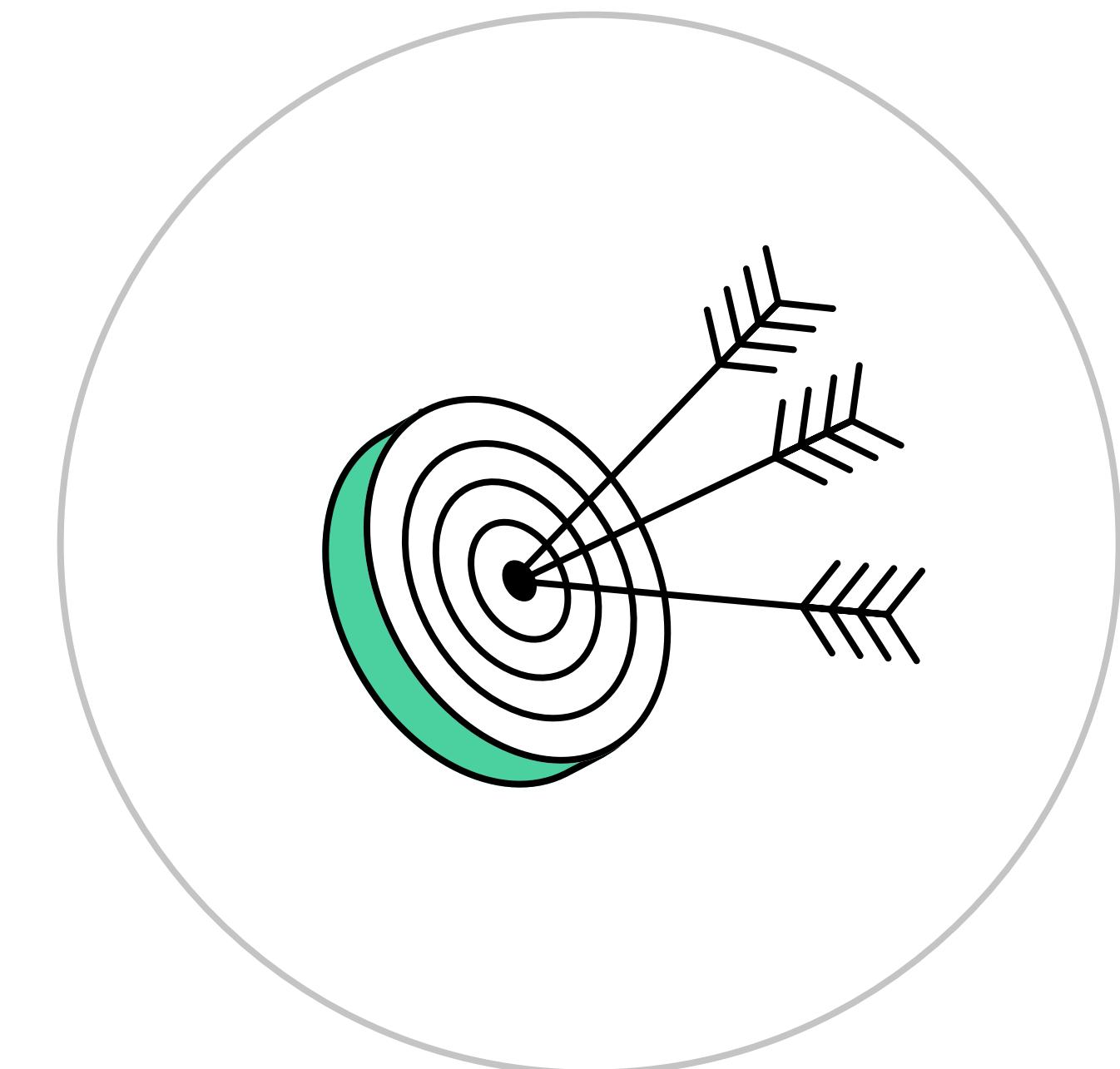


Object, Reflexion, Proxy



Цели занятия

- Углубить знания об объектах и научиться добавлять и удалять свойства
- Получить практические навыки использования прототипов и цепочек прототипов
- Научиться перебирать свойства и сравнивать объекты
- Познакомиться с шаблоном проектирования Proxy & Reflect



План занятия

- 1 Объекты
- 2 Свойства объекта
- 3 Прототипы и цепочки прототипов
- 4 Object
- 5 Перебор свойств
- 6 Proxy & Reflect



Объекты

Объекты

В JavaScript объекты представляют из себя набор свойств (пар «ключ-значение»).

Объекты в JavaScript и программировании в целом – это основные средства для организации данных и функций в программе. Они позволяют создавать комплексные структуры данных и управлять их поведением.

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']);
```

Обращение через точку более простой и удобный способ доступа к свойствам объекта, если известно имя свойства.

Если оно неизвестно и хранится в переменной, то необходимо использовать обращение через квадратные скобки. Этот способ пригодится, если имя свойства содержит специальные символы или начинается с цифры

Извлечение свойств

ES6 предоставляет удобный способ извлечения свойств с помощью **Object Destructuring**:

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

Добавление и удаление свойств

В JavaScript мы в любой момент можем добавить объекту новое свойство или 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 никогда в объекте не было, то попытка доступа закончится тем, что мы получим `undefined`:

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

Проверка на undefined

Давайте подумаем, чем плох следующий код:

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

Проверка на undefined

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

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

Чтобы извлечь свойства из объекта, который является свойством другого объекта можно использовать Object Destructuring.

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

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 Destructuring**:

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

В `rest` будет:

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

Rest

Это даёт возможность создать **Shallow Copy** (поверхностную копию) для объектов:

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

Также мы сможем объединить несколько объектов в один:

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

```
1  'use strict';
2  function App()
3
4  App.prototype.demo = function() {
5    console.log(this);
6  }
7
8  const app = new App();
9  const result = new app.demo(); // OK
10
11 // реализация на классах
12
13 class App {
14   demo() {
15     console.log(this);
16   }
17 }
18
19 const app = new App();
20 const result = new app.demo(); // TypeError
```

Задача

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

- ответственный
- приоритет
- категория

Т. е. у одних объектов такие свойства могут быть, а других – нет.

Задача: 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!
```

Хотя в объекте `user` нет свойства с таким именем, результатом будет `true`. Дело в том, что свойство `toString` унаследовано от прототипа объекта `user`, который является экземпляром стандартного объекта `Object`. Свойство `toString` – встроенный в объект `Object` метод, и поэтому оно доступно для всех объектов, созданных на основе этого прототипа

Прототипы и цепочки прототипов

Прототипы

JavaScript – 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 объекта, то JavaScript сначала ищет это свойство в объекте, затем 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 мы создаём 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__);
```

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

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

Object

Object

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

В частности 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 стоит использовать метод `setPrototypeOf` в production code. Мы его применяем 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

Перебор свойств

Перебор свойств

Мы рассмотрели оператор `in`, который позволяет проверять наличие свойства в объекте, включая цепочку прототипа. Давайте теперь обсудим другой оператор – `for...in`:

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



The screenshot shows a browser developer tools console window. The title bar has three small circles. The console area contains the following text:

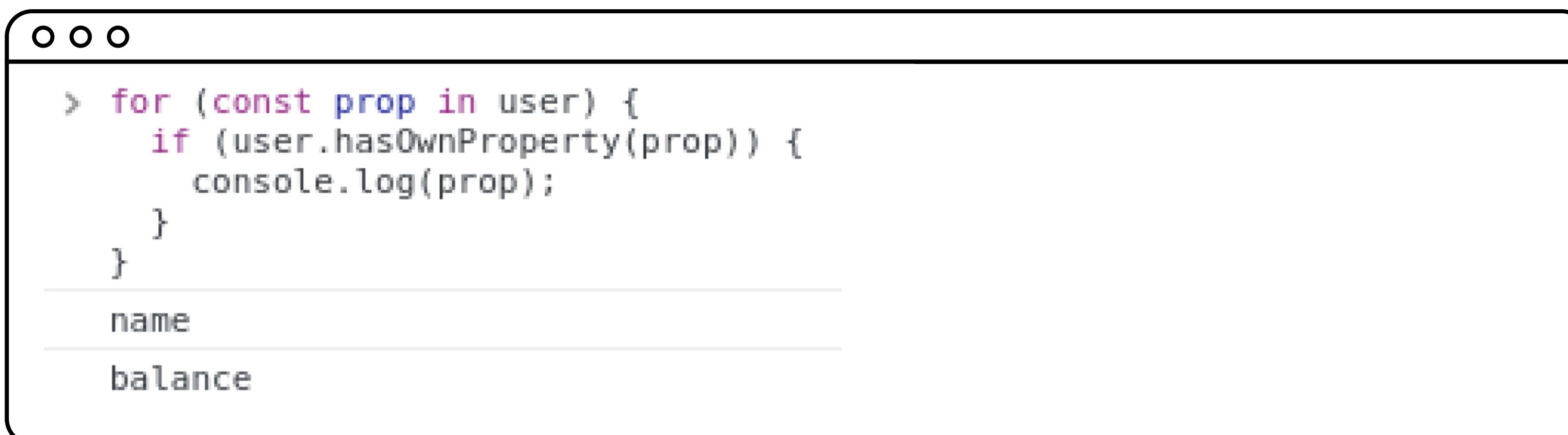
```
o o o  
> for (const prop in user) {  
  console.log(prop);  
}  
name  
balance  
id
```

The words "name", "balance", and "id" are displayed in the console, indicating they are properties of the "user" object.

HasOwnProperty

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 | }
```



The screenshot shows a browser developer tools console window. At the top, there are three small circular icons. Below them, the console prompt starts with three greater-than signs (">"). The code from the previous block is pasted into the console. When the code is run, it logs two properties to the console: "name" and "balance".

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

Можно сделать проще

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

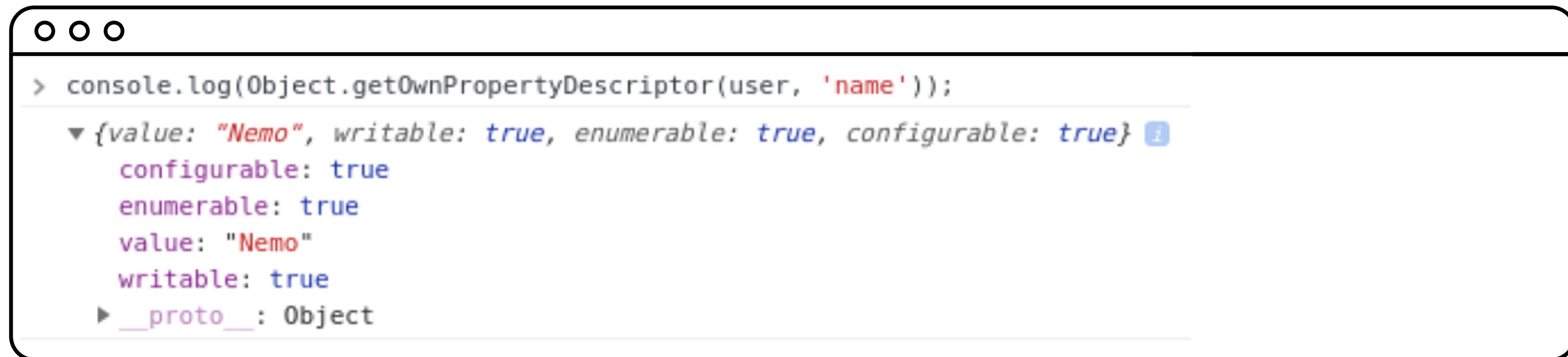
При создании свойства в объекте мы можем определить ряд характеристик (дескриптор), которые определяют поведение этого свойства. Вот что по этому поводу написано в [MDN](#):

- **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'));
```



The screenshot shows a browser's developer tools console window with three circular icons in the top-left corner. The console output is as follows:

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

The output shows the descriptor object for the 'name' property of the 'user' object. It has four properties: 'value' (set to 'Nemo'), 'writable' (set to true), 'enumerable' (set to true), and 'configurable' (set to true). The 'configurable' property has an info icon next to it. The 'value', 'writable', and 'enumerable' properties are colored blue, while 'configurable' is purple. The 'user' object itself is represented by a light gray box with a black border.

Object.getOwnPropertyDescriptor

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

```
ooo

> 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, а изменится существующее.

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

Q & A

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

Ответ: 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.

ИТОГИ

Q: Как перебрать все свойства объекта?

A: Зависит от ваших целей. Если вам необходимы:

- все перечисляемые, включая цепочку прототипов, то через `for..in`.
- все перечисляемые собственные, то `for..in + hasOwnProperty` либо `Object.entries`

Q: Можно ли в объект добавить свойство или удалить его из него?

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 понять, как устроен сам язык и как работают библиотеки. Например, библиотека [Immutable.js](#) активно использует дескрипторы свойств.

Во-вторых, это часто спрашивают на собеседованиях, проверяя то, насколько вы погружались в сам язык.

Поэтому мы рассмотрим ещё ряд тем подобного рода

ToString

Когда мы пытаемся использовать объект в «строковом контексте», вызывается метод `toString`, который определён в цепочке прототипов.

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

Если мы напишем свой метод `toString`, то по правилам JavaScript сначала будет искать это свойство в нашем объекте, и только если не найдёт – начнёт 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

Описанный ниже код не будет работать, потому что стрелочные функции не создают свою локальную область видимости `this`. Вместо этого они наследуют `this` из окружающего контекста.

В этом случае, окружающий контекст – это глобальный контекст выполнения кода, поэтому `this` будет ссылаться на глобальный объект: в браузере это `window`, в Node.js – `global`

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

Как правильно объявлять методы

Попробуем создать новый объект, в котором сразу в лiteralной форме пропишем 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 или 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 };
```

Важно: стоит использовать более новый синтаксис, если есть такая возможность

Задача

Необходимо сравнить два объекта, например, при поиске или сортировке. Варианты решения:

- сравнение свойств
- `valueOf`

Рассмотрим вариант `valueOf`

ValueOf

Метод прототипа, который вызывается при преобразовании объекта в примитивный тип (не в строковый контекст).

Пример:

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

Переопределение `valueOf` позволяет задать собственные правила

Как сравнивать объекты на равенство

Это можно сделать только через сравнение полей.

Если мы хотим сравнивать в контексте приведения
к примитивным типам либо приводить к ним,
то переопределим `valueOf`.

`Object.is`

Object.is

В описании работы библиотек часто встречается этот метод.
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

Необходимо знать, как он работает. Возвращается `Object.is(a, b) === true` в случаях:

- `a` и `b` равны
- `a` и `b` равны `null`
- `a` и `b` равны `true` или `false`
- `a` и `b` — строки с одинаковым содержимым и длиной
- `a` и `b` указывают на один объект
- `a` и `b` — числа и равны `+0` или `-0`
- `a` и `b` — числа и равны `NaN`
- `a` и `b` — числа и равны одному и тому же числу, но не `+/-0` или `NaN`

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

Proxy & Reflect

Proxy

Proxy — широко распространённый метод в разработке, который позволяет подкладывать объект, перехватывающий вызовы, к оригинальному объекту.

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

Общая схема выглядит так:

вызывающий код → Proxy → оригинальный объект (target)

Т. е. Proxy может перехватывать большинство вызовов и либо перенаправлять их к 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`

Extends

Пример:

```
○○○
> const user = {name: 'Nemo', balance: 10000};
< undefined
> const proxy = new Proxy(user, {
  get(target, key, receiver) {
    console.log(target); console.log(key);
    return 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** – оригиналъный объект user
- **receiver** – контекст вызова. This – сам Proxy либо объект, у которого он в цепочке прототипов

Reflection

Рефлексия — это возможность анализировать приложение, например объекты, во время исполнения, а не написания.

В JavaScript 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 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 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)`.

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

```
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          return 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 этого конспекта

Зачем применять Proxy + Reflect

- Для отладки кода — чтобы посмотреть, какие функции, включая `Object.`, библиотеки вызывают на вашем объекте и что может пойти не так
- Написания библиотек — возможность перехватывать обращения к методам позволяет встраивать собственную логику, так называемое аспектно-ориентированное программирование, внедряя логирование, контроль доступа и т. д.