нетология

Объекты

Владимир
Чебукин

Владимир Чебукин

Frontend-разработчик

Владимр Чебукин

План занятия

1. Вспомним то, что мы уже знаем
2. Удаление свойств и объектов
3. Контекст выполнения
4. Функции-конструкторы расширенно
5. Свойство prototype функции-конструктора
6. Конструктор Date и его методы
7. Методы объекта Math
8. Итоги

Вспомним то, что мы уже знаем

Зачем нужны объекты?

Объекты позволяют **удобно организовать хранение** информации.

Объект можно создать по-разному:

```
let person = new Object(); // устаревший способ
let person = {}; // более современный способ создаем пустой объект

// Затем присвоить свойства
person.firstName = "Иван";
person.lastName = "Орлов";
person.age = 45;

// форма ниже называется литерал объекта
let person = {
  firstName: "Иван",
  lastName: "Орлов",
  age: 45,
}
```

Зачем нужны объекты?

Такой подход **нагляднее**, чем:

```
let person = ["Иван", "Орлов", 45]; // что такое 45?
```

На самом деле, массивы и функции — тоже объекты

В JavaScript почти всё является **объектом**.

Массивы:

```
let data = new Array("Иннокентий", "Ильдар", "Ирина");

function sum(a, b) {
  return a + b;
}

console.log(typeof data); // object
console.log(typeof sum); // function
```

Заметьте, что *typeof sum === "function"*, но при этом отдельного типа *function* нет.

Это тот же объект, у которого есть **свойства**

Свойства. Кастомные свойства

У **объектов есть свойства**. Вы можете задавать объекту абсолютно произвольные свойства с произвольными значениями

Код на следующей странице

Свойства. Кастомные свойства

```
let customProperty = "isCat";
let person = {
  firstName: "Иван",
  // свойства можно задавать и в кавычках
  lastName: "Орлов",
  // кавычки удобны для задания специфических значений
  "font-size": "20px",
  // ES6+, customProperty может быть любым JS-выражением
  [customProperty]: false, // создаст свойство isCat со значением false
};

// можно задавать свойства после создания объекта
person.fatherName = "Борис";
person["patronym"] = "Борисович";

let newProperty = "gender";
// в квадратные скобки можно поместить любое JS-выражение
person[newProperty] = "male"; // создаст свойство gender со значением male
```

ES6+. Сокращённое задание свойств

Можно задавать свойства на основе **имени переменной**:

```
let firstName = "Иван";

let person = {
  firstName: firstName,
}
```

Полностью аналогично:

```
let firstName = "Иван";

let person = {
  firstName,
}
```

Когда вы видите конструкцию *property: property,* почти во всех случаях можно заменить её на *property*

Чтение свойств

Чтение свойств аналогично их заданию:

```
let person = {
  firstName: "Иван",
  lastName: "Орлов",
  age: 45,
};

console.log(person.firstName); // Иван
console.log(person["lastName"]); // Орлов

let myProperty = "age";
console.log(person[myProperty]); // 45
```

Несуществующие свойства

Доступ к любому несуществующему свойству даёт *undefined*:

```
let person = {
  firstName: "Иван",
  lastName: "Орлов",
  age: 45,
}

console.log(person.fatherName); // undefined
```

Объекты являются значениями по ссылке

Как вы знаете из лекции *«Функции»,* в отличие от примитивных типов данных при **присваивании объекта** копируется не значение, а **ссылка** на этот объект.

Примитивы:

```
let x = 6,
  y = x;

y = 9;

console.log(x, y);
// 6, 9
```

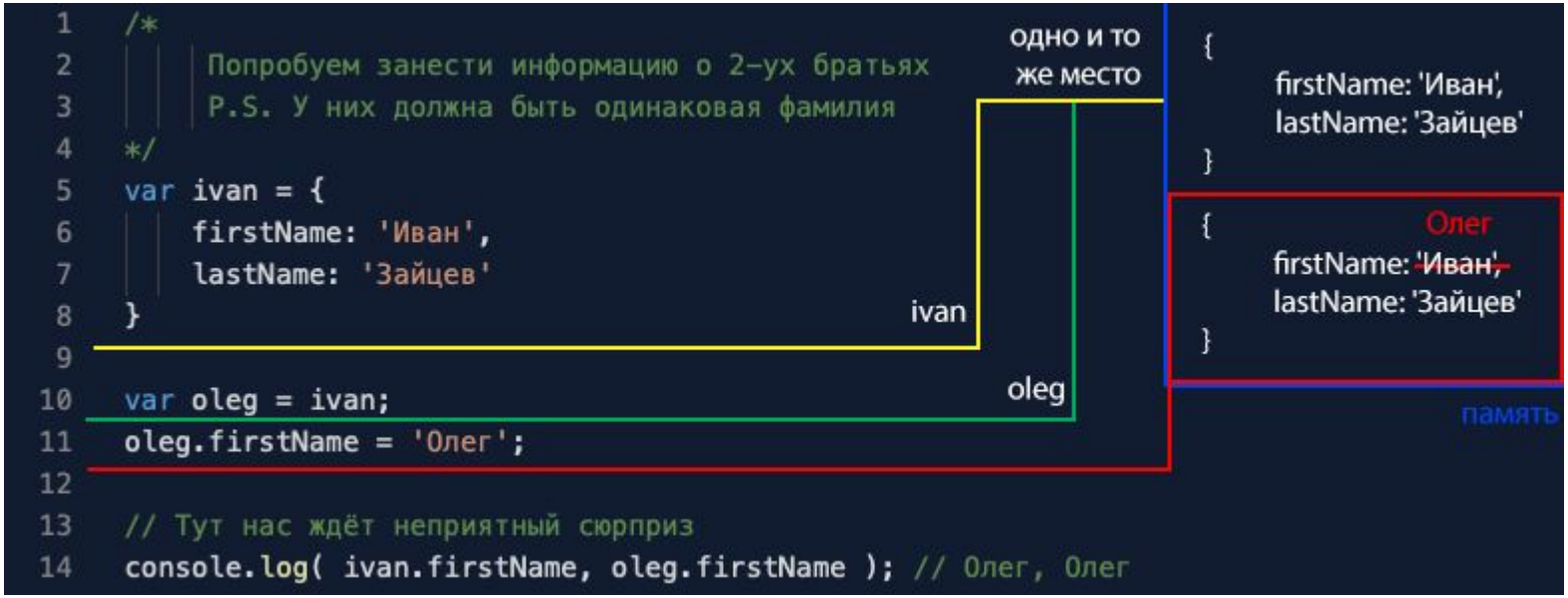
Объекты:

```
let ivan = {
  firstName: "Иван",
  lastName: "Зайцев",
};

let oleg = ivan;
oleg.firstName = "Олег";

console.log(ivan.firstName, oleg.firstName);
// Что будет выведено?
```

Пояснение


/*
Попробуем занести информацию о 2-ух братьях
P.S. У них должна быть одинаковая фамилия
*/
var ivan = {
firstName: 'Иван',
lastName: 'Зайцев'
}
var oleg = ivan;
oleg.firstName = 'Олег';
// Тут нас ждёт неприятный сюрприз
console.log(ivan.firstName, oleg.firstName); // Олег, Олег
одно и то же место
ivan
oleg
{
firstName: 'Иван',
lastName: 'Зайцев'
}
Олег
{
firstName: Иван,
lastName: 'Зайцев'
}
память

Обход свойств

Один из способов **обхода всех свойств объекта** — использование конструкции *for in*:

```
let ivan = {
  firstName: "Иван",
  lastName: "Зайцев",
};

for (let prop in ivan) {
  let value = ivan[prop];
  console.log(`Свойство ${prop}, значение: ${value}`);
}
```

Обход свойств

Один из способов **обхода всех свойств объекта** — использование конструкции *for in*:

```
let ivan = {
  firstName: "Иван",
  lastName: "Зайцев",
};

for (let prop in ivan) {
  let value = ivan[prop];
  console.log(`Свойство ${prop}, значение: ${value}`);
}
```

Результат работы:

```
Свойство firstName, значение: Иван
Свойство lastName, значение: Зайцев
```

Обойти свойства также можно с помощью методов *Object.keys* и *Object.entries* и последующей итерации с помощью цикла. Об этом расскажем далее

Доступ к несуществующим свойствам

Возможна ситуация, когда мы заранее не знаем, существует **свойство** или нет:

```
let user = {};
console.log(user.name.first); // TypeError: Cannot read property 'first' of undefined
```

Раньше это решалось так:

```
if (user && user.name && user.name.first) {
    console.log(user.name.first);
}
```

Новая возможность *optional chaining* доступна в ECMAScript 2020:

```
if (user?.name?.first) {
    console.log(user.name.first);
}
```

Удаление свойств и объектов

Как происходит удаление объектов в javascript

Во время выполнения скрипта для каждого **примитива** или **объекта** выделяется определённый **участок памяти**.

Память не бесконечна, поэтому её требуется периодически очищать от мусора — **неиспользуемых** значений переменных, объектов и их свойств. За этим следит **сборщик мусора** — алгоритм, очищающий память.

Как понять, можно ли удалить какое-то значение? Это просто. Значение считается неиспользуемым, если на него **не ведёт никакая ссылка**

delete

Оператор *delete* позволяет удалять свойства объектов. Элемент массива — это тоже свойство объекта: *индекс — имя свойства.*

С помощью *delete* можно удалить **только** свойство объекта и нельзя удалить переменные, *объявленные через var и let*:

```
arr = {
  0: 'some',
  1: 'value',
  length:2;
}
```

Синтаксис:

```
delete object.property;
delete object['property'];
delete array[index];
```

Delete имеет свои особенности

1

delete возвращает *false* только в том случае, если свойство существует, но не может быть удалено, и *true* — в любых других случаях:

```
let anybodyObject = {
  first: 10,
  second: 20,
};

delete anybodyObject.first; // true
delete anybodyObject.third; // true
console.log(anybodyObject); // {second: 20}
```

Delete имеет свои особенности

2

При удалении элемента массива в массиве сохраняется пустое место (*empty*) от этого элемента. Длина массива при этом не изменится:

```
let array = ["first", "second", "third"];
delete array[2]; //true
console.log(array); // ["first", "second", empty]
```

Delete имеет свои особенности

3

delete не изменяет прототип объекта. Существуют свойства, которые нельзя удалить:

```
f = [1, 2, "third"];
delete f.length; // false;
```

Возврат объектов в функциях

Эта информация будет полезна при решении ДЗ!

Напишем функцию, которая выводит информацию о сотруднике компании:

```
function getProfile(firstName, lastName, birthYear, jobYear) {
  // Текущий год
  let year = new Date().getFullYear();
  return {
    firstName,
    lastName,
    name: firstName + " " + lastName,
    birthYear,
    age: year - birthYear,
    jobYear,
    seniority: year - jobYear,
  }
}

console.log(getProfile("Максим", "Иванов", 1980, 2000));
```

Возврат объектов в функциях

Эта информация будет полезна при решении ДЗ!

Напишем функцию, которая выводит информацию о сотруднике компании:

```
function getProfile(firstName, lastName, birthYear, jobYear) {
  // Текущий год
  let year = new Date().getFullYear();
  return {
    firstName,
    lastName,
    name: firstName + " " + lastName,
    birthYear,
    age: year - birthYear,
    jobYear,
    seniority: year - jobYear,
  }
}

console.log(getProfile("Максим", "Иванов", 1980, 2000));
```

У нас получилась **функция-конструктор**:

```
{ firstName: 'Максим',
  lastName: 'Иванов',
  name: 'Максим Иванов',
  birthYear: 1980,
  age: 38,
  jobYear: 2000,
  seniority: 18 }
```

Методы

Если в свойстве объекта значением будет функция, такое свойство называется **методом**:

```
let person = {
  firstName: "Иван",
  showName: function () {
    console.log(`Имя: ${this.firstName}`)
  },
  // Начиная с ES 5 то же самое можно записать следующим образом
  showName2() {
    console.log(`Имя: ${this.firstName}`);
  },
}

// вызов метода
person.showName(); // Имя: Иван
const func = person.showName;
// func указывает на уже созданную функцию (не копирование)
func(); // Имя: undefined
```

Методы

Если в свойстве объекта значением будет функция, такое свойство называется **методом**.

Когда функция **вызывается** как метод объекта, а *не сама по себе*, *this* указывает на объект перед точкой*: правило — объект до точки*. В другом случае в зависимости от режима (*use strict*) либо на глобальный объект (*window*, *global*), либо *undefined*

Отличие стрелочных функций от обычных

Рассмотрим отличие стрелочных функций от обычных:

```
let person = {
  firstName: "Иван",
  printName: function () {
    return this.firstName
  },
  printNameArrow: () => this.firstName,
}
console.log(person.printName()); // "Иван"
console.log(person.printNameArrow()); // "undefined"
```

Отличие стрелочных функций от обычных

```
let person = {
  firstName: "Иван",
  printName: function () {
    return this.firstName
  },
  printNameArrow: () => this.firstName,
}
console.log(person.printName()); // "Иван"
console.log(person.printNameArrow()); // "undefined"
```

Вывод: стрелочные функции особенные. У них нет своего собственного *this*.

Если мы используем *this* внутри стрелочной функции, то его значение берётся из внешней нормальной функции

Функции-конструкторы расширенно

Функции-конструкторы расширенно

Мы уже знакомились с созданием объектов с помощью функции-конструктора:

```
function Car(engine, number) {
  this.engine = engine;
}

const car1 = new Car("V8", "E234KX");
const car2 = new Car("V16", "032100");
console.log(car1.engine); // V8
console.log(car2.engine); // V16
```

Вспомним, что делает оператор **new**

Оператор *new*

Позволяет создавать объекты **через вызов функций**.

Особенности работы функций, вызванных через оператор **new**:

- создается новый пустой **объект**
- ключевое слово *this* получает ссылку на этот объект
- **функция** выполняется
- возвращается *this* без явного указания

Свойство prototype функции-конструктора

Пример

Можно реализовать метод, который смогут использовать все инстансы — *экземпляры*, чтобы не прописывать метод в каждом из объектов:

```
function User(name, lastName) {
  this.name = name;
  this.lastName = lastName;
}
User.prototype.getFullName = function () {
  console.log("Полное имя: " + this.name + " " +
this.lastName);
}

const user1 = new User("Иван", "Иванович");
const user2 = new User("Петр", "Пустота");
console.log(user1.name); // Иван
console.log(user2.name); // Петр

user1.getFullName(); // ?
user2.getFullName();
```

Для этого мы должны поместить метод (*функцию*) в свойство *prototype* функции-конструктора.

Обратите внимание: *user1* и *user2* имеют доступ к методу *getFullName*

Прототип объекта

Сейчас данный функционал в 99% случаев реализуется с помощью **классов**, в которые встроено удобное наследование и другие полезные фишки. О классах расскажем в следующей лекции

Если всё же вам хочется погрузиться в прототипы, начать можно отсюда:

https://learn.javascript.ru/native-prototypes

Object.hasOwnProperty()

Для примера всё же создадим объект *Tiger*, прототипом которого будет *Predator*:

```
const Predator = {
  food: "meat",
}
function Tiger(name) {
  this.name = name
}

Tiger.prototype = Predator; // определяем прототип

const tiger = new Tiger("Vasya");

tiger.hasOwnProperty("name"); // true
tiger.hasOwnProperty("food"); // false

"name" in tiger; // true
"food" in tiger; // true
```

Метод *hasOwnProperty()* определяет, **содержит ли** объект указанное свойство в качестве собственного свойства объекта.

В отличие от оператора *in*, он не проверяет наличие указанного свойства по цепочке прототипов

Object.hasOwnProperty()

Для примера всё же создадим объект *Tiger*, прототипом которого будет *Predator*:

```
const Predator = {
  food: "meat",
}
function Tiger(name) {
  this.name = name
}

Tiger.prototype = Predator; // определяем прототип

const tiger = new Tiger("Vasya");

tiger.hasOwnProperty("name"); // true
tiger.hasOwnProperty("food"); // false

"name" in tiger; // true
"food" in tiger; // true
```

Тут одно простое, но важное правило. Поиск вверх по цепочке начинается со свойства *prototype* конструктора и далее до *Object*

Object.create(prototype)

Создаёт новый объект с указанным объектом прототипа (*prototype*):

```
const tiger = Object.create(predator);
tiger.food; // meat
```

Близко к предыдущему примеру, но не полностью аналогично

Object.assign(target,...sources)

Создает **новый объект** путём копирования значений всех собственных перечисляемых свойств из одного или более исходных объектов в целевой объект. В новом стандарте есть аналог — `...`:

```
const person = {
  name: "Frederic",
}

const account = {
  balance: 14000,
}

const info1 = Object.assign({}, person, account);
info1.name; // 'Frederic'
info1.balance; // '14000

// современный способ
const info2 = { ...person };
info2.name; // 'Frederic'

// Еще способ (не копирует методы)
let newObj = JSON.parse(JSON.stringify(person));

// Еще можно циклом for in
```

Глубокое клонирование

В предыдущем примере мы предполагали, что все свойства объекта **хранят примитивные значения**. Но свойства могут быть **ссылками** на другие объекты. Что с ними делать?

```
let car1 = {
  name: "Ford",
  engine: {
    volume: 2.0,
    eco: 5,
  },
}

let car2 = Object.assign({}, user); // клон car1

car1.engine === car2.engine;
// true, один и тот же объект, а нам хотелось бы клон
```

Решение

Чтобы исправить это, следует в цикле клонирования проверять, не является ли значение *car1[key]* объектом, и если это так — скопировать и его структуру тоже. Это называется **глубокое клонирование**.

Мы можем реализовать глубокое клонирование, используя **рекурсию**. Или, чтобы не изобретать велосипед, использовать готовую реализацию — метод ***_.cloneDeep(obj)*** из JavaScript-библиотеки lodash.

Object.keys(obj) | *Object.values(obj)*

- *keys* — возвращает массив, содержащий имена всех собственных перечисляемых свойств переданного объекта
- *values* — возвращает массив значений перечисляемых свойств объекта в том же порядке, что и цикл *for...in*

```
Object.keys(info); // ["name", "balance"]

Object.values(info); // ["Frederic", 14000]
```

Object.entries(obj)

Метод **возвращает** массив собственных перечисляемых свойств указанного объекта в формате *[key, value]* в том же порядке, что и в цикле *for...in*. *Разница в том, что for-in также перечисляет свойства из цепочки прототипов:*

```
Object.entries(info); // [ ["name", "Frederic"], ["balance", 14000] ]
```

1. Он не идёт вглубь по прототипам!
2. Удобно итерировать.

Object.entries(obj)

Ниже приведены примеры итерации по свойствам объекта:

```
for (const [key, value] of Object.entries(obj)) {
  console.log(`${key} ${value}`);
}

Object.entries(obj).forEach(([key, value]) => {
  console.log(`${key} ${value}`);
});

for (const key of Object.keys(obj)) {
  console.log(`${key} ${obj[key]}`);
}

let keys = Object.keys(obj);
for (let i = 0; i < keys.length; i++) {
  console.log(`${keys[i]} ${obj[keys[i]]}`);
}
```

Встроенный объект Date и его методы

Работа с датами

Для работы с **датами** в JS предусмотрен **глобальный объект** *Date*. Он сложный, противоречивый и местами опасный.

Обратим внимание, что в документации MDN (https://developer.mozilla.org/ru/docs/Web/JavaScript/) такие объекты в разных местах называются конструктором, глобальным объектом, встроенным глобальным объектом. Все три названия верны, так как *Date* может использовать по-разному:

```
let today = new Date(1995, 11, 17) // используется как конструктор
let start = Date.now() // используется как объект
```

Конструктор *Date*

Дата и время представлены в JavaScript одним объектом *Date*:

```
new Date(); // "Fri Nov 23 2018 01:03:26 GMT+0300 (Москва, стандартное время)"
new Date(2018, 0, 1, 12, 00);
// "Mon Jan 01 2018 12:00:00 GMT+0300 (Москва, стандартное время)"
```

- Отсчет месяцев начинается с нуля.
- Отсчет дней недели *для getDay()* тоже начинается с нуля. *Нулём является воскресенье.*

Стандартные форматы дат и времени

1

ISO 86001 — в виде строки:

```
date.toString();
// "Thu Nov 22 2018 21:48:54 GMT+0300 (Москва, стандартное время)"
```

Удобно для сортировок и человеко-читаемо.

2

Unix Timestamp — в виде числа:

```
Number(date); // 1542912588582
```

Удобно находить разницу между временными метками

Установка компонент дат

```
date.setFullYear(year [, month, date]);
date.setMonth(month [, date]);
date.setDate(date);
date.setHours(hour [, min, sec, ms]);
date.setMinutes(min [, sec, ms]);
date.setSeconds(sec [, ms]);
date.setMilliseconds(ms);
...
```

Дополнительная информация:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date

Получение компонент дат

```
const date = new Date();
date.getFullYear(); // 2018
date.getMonth(); // 10
date.getDate(); // 22
date.getHours(); // 21
date.getMinutes(); // 32
date.getSeconds(); // 41
date.getMilliseconds(); // 122
...
```

Дополнительная информация:

https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Date

Метод *Date.now()*

Метод *Date.now()* возвращает дату сразу в виде миллисекунд:

```
Date.now(); // 1542912986688
```

Такого результата можно добиться:

```
Number(new Date()); // 1542912986688
```

Автоисправление даты

Неправильные компоненты даты автоматически распределяются по остальным:

```
new Date(2018, 12, 40) // Sat Feb 09 2019 00:00:00 GMT+0300
```

Внимание! Будьте осторожны с этой особенностью *Date*

Intl

Объект, позволяющий **интернационализировать** даты:

```
const date = new Date();

const formatter = new Intl.DateTimeFormat("ru");
formatter.format(date); // "23.11.2018"
```

Методы объекта Math

Пример

Предположим, вы реализуете интерфейс, и вам понадобилось **сгенерировать случайное число** или округлить его.

Для подобных случаев в JS предусмотрен объект *Math*, в котором есть все часто используемые функции, касающиеся **математических выражений**

Объект *Math*

Math — глобальный объект, помогающий в математических вычислениях.

Имеет **свойства**, **константы**:

```
Math.PI; // 3.141592653589793
Math.E; // 2.718281828459045
```

И **методы**:

```
Math.random(); // Случайное число от 0 до 1
Math.max(12, 13, 432, -1); // 432
Math.min(12, 13, 432, -1); // -1
```

Округление чисел

```
Math.floor(2.6); // в меньшую сторону
Math.round(2.6); // до ближайшего целого
Math.ceil(2.1); // в большую сторону
```

Модуль числа

Для нахождения модуля числа используют *Math.abs(value)*:

```
Math.abs(-2); // 2
Math.abs(2); // 2
```

Number.toFixed()

Ещё один очень полезный метод для работы с числами. Обратите внимание, что он **возвращает строку**:

```
var numObj = 12345.6789;
numObj.toFixed(6);
numObj.toFixed(); // '12346': обратите внимание на округление, дробной части нет
numObj.toFixed(1); // '12345.7': обратите внимание на округление
numObj.toFixed(6); // '12345.678900': обратите внимание на дополнение нулями
```

Number.toFixed()

Позволяет в некоторых случаях избавиться от проблем **с числами с плавающей точкой** в js:

```
console.log(0.1 + 0.2 === 0.3); // false
console.log((0.1 + 0.2).toFixed(2) === (0.3).toFixed(2)); // true
```

Итоги

Чему мы научились?

- **Узнали** про основы работы с объектами: задание, чтение, обход их свойств
- **Познакомились** с контекстом *this*
- **Разобрали** понятие конструктора и оператор *new*
- **Узнали,** как добавить общие методы объектов (*prototype*)
- **Познакомились** с методами объекта *Object*
- **Познакомились** с объектом *Math*, помогающим в математических вычислениях, и объектом *Date*, облегчающим манипуляции с датами

Домашнее задание

Давайте посмотрим ваше домашнее задание:

- **вопросы** по домашней работе задаём в группе Slack;
- задачи можно сдавать **по частям**
- зачёт по домашней работе проставляется после того, как приняты все **3 задачи**

нетология

Задавайте вопросы и пишите отзыв о лекции!

Владимир Чебукин

Владимир Чебукин