**Лекция 15.**

**JavaScript: ES6 ­– Объекты и прототипы, классы, наследование.**

[Объекты и прототипы 3](#_Toc519459427)

[Короткое свойство 3](#_Toc519459428)

[Вычисляемые свойства 3](#_Toc519459429)

[Геттер-сеттер для прототипа 4](#_Toc519459430)

[Object.assign 4](#_Toc519459431)

[Object.is(value1, value2) 5](#_Toc519459432)

[Методы объекта 5](#_Toc519459433)

[super 6](#_Toc519459434)

[Классы 10](#_Toc519459435)

[Class 10](#_Toc519459436)

[Class Expression 11](#_Toc519459437)

[Геттеры, сеттеры и вычисляемые свойства 12](#_Toc519459438)

[Статические свойства 13](#_Toc519459439)

[Наследование 14](#_Toc519459440)

[Дополнительные материалы 17](#_Toc519459441)

[Задание для закрепления материала 18](#_Toc519459442)

# Объекты и прототипы

## Короткое свойство

Зачастую у нас есть переменные, например, name и isAdmin, и мы хотим использовать их в объекте.

При объявлении объекта в этом случае достаточно указать только имя свойства, а значение будет взято из переменной с аналогичным именем.

Например:

'use strict';

let name = "Вася";

let isAdmin = true;

let user = {

name,

isAdmin

};

console.log( JSON.stringify(user) ); // {"name": "Вася", "isAdmin": true}

## Вычисляемые свойства

В качестве имени свойства можно использовать выражение, например:

'use strict';

let propName = "firstName";

let user = {

[propName]: "Вася"

};

console.log( user.firstName ); // Вася

Или даже так:

'use strict';

let a = "Мой ";

let b = "Зелёный ";

let c = "Крокодил";

let user = {

[(a + b + c).toLowerCase()]: "Гена"

};

console.log( user["мой зелёный крокодил"] ); // Гена

## Геттер-сеттер для прототипа

В ES5 для прототипа был метод-геттер:

* Object.getPrototypeOf(obj) - возвращает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта.

В ES-2015 также добавился сеттер:

* Object.setPrototypeOf(obj, newProto) - устанавливает прототип (то есть, внутреннее свойство [[Prototype]]) указанного объекта в другой объект или null.

А также «узаконено» свойство \_\_proto\_\_, которое даёт прямой доступ к прототипу. Его, в качестве «нестандартного», но удобного способа работы с прототипом, реализовали почти все браузеры (кроме IE10-), так что было принято решение добавить его в стандарт.

## Object.assign

Функция Object.assign получает список объектов и копирует в первый target свойства из остальных.

Синтаксис:

Object.assign(target, src1, src2...)

При этом последующие свойства перезаписывают предыдущие.

Например:

'use strict';

let user = { name: "Вася" };

let visitor = { isAdmin: false, visits: true };

let admin = { isAdmin: true };

Object.assign(user, visitor, admin);

console.log( JSON.stringify(user) ); // name: Вася, visits: true, isAdmin: true

Его также можно использовать для 1-уровневого клонирования объекта:

'use strict';

let user = { name: "Вася", isAdmin: false };

// clone = пустой объект + все свойства user

let clone = Object.assign({}, user);

## Object.is(value1, value2)

Это новая функция для проверки равенства значений. Возвращает true, если значения value1 и value2 равны, иначе false. Она похожа на обычное строгое равенство ===, но есть отличия:

// Сравнение +0 и -0

console.log( Object.is(+0, -0)); // false

console.log( +0 === -0 ); // true

// Сравнение с NaN

console.log( Object.is(NaN, NaN) ); // true

console.log( NaN === NaN ); // false

Отличия эти в большинстве ситуаций некритичны, так что не похоже, чтобы эта функция вытеснила обычную проверку ===. Что интересно – этот алгоритм сравнения, который называется SameValue, применяется во внутренних реализациях различных методов современного стандарта.

## Методы объекта

Долгое время в JavaScript термин «метод объекта» был просто альтернативным названием для свойства-функции.

Теперь это уже не так. Добавлены именно «методы объекта», которые, по сути, являются свойствами-функциями, привязанными к объекту.

Их особенности:

1. Более короткий синтаксис объявления.
2. Наличие в методах специального внутреннего свойства [[HomeObject]] («домашний объект»), ссылающегося на объект, которому метод принадлежит.

Для объявления метода вместо записи "prop: function() {…}" нужно написать просто "prop() { … }".

Например:

'use strict';

let name = "Вася";

let user = {

name,

// вместо "sayHi: function() {...}" пишем "sayHi() {...}"

sayHi() {

console.log(this.name);

}

};

user.sayHi(); // Вася

Как видно, для создания метода нужно писать меньше букв. Что же касается вызова – он ничем не отличается от обычной функции. На данном этапе можно считать, что «метод» – это просто сокращённый синтаксис для свойства-функции.

Также методами станут объявления геттеров get prop() и сеттеров set prop():

'use strict';

let name = "Вася", surname="Петров";

let user = {

name,

surname,

get fullName() {

return `${name} ${surname}`;

}

};

console.log( user.fullName ); // Вася Петров

Можно задать и метод с вычисляемым названием:

'use strict';

let methodName = "getFirstName";

let user = {

// в квадратных скобках может быть любое выражение,

// которое должно вернуть название метода

[methodName]() { // вместо [methodName]: function() {

return "Вася";

}

};

console.log( user.getFirstName() ); // Вася

## super

В ES-2015 появилось новое ключевое слово super. Оно предназначено только для использования в методах объекта.

Вызов super.parentProperty позволяет из метода объекта получить свойство его прототипа.

'use strict';

let animal = {

walk() {

console.log("I'm walking");

}

};

let rabbit = {

\_\_proto\_\_: animal,

walk() {

super.walk(); // I'm walking

}

};

rabbit.walk();

Как правило, это используется в классах, которые мы рассмотрим в следующем разделе, но важно понимать, что «классы» здесь на самом деле ни при чём. Свойство super работает через прототип, на уровне методов объекта.

При обращении через super используется [[HomeObject]] текущего метода, и от него берётся \_\_proto\_\_. Поэтому super работает только внутри методов.

В частности, если переписать этот код, оформив rabbit.walk как обычное свойство-функцию, то будет ошибка:

'use strict';

let animal = {

walk() {

console.log("I'm walking");

}

};

let rabbit = {

\_\_proto\_\_: animal,

walk: function() { // Надо: walk() {

super.walk(); // Будет ошибка!

}

};

rabbit.walk();

Ошибка возникнет, так как rabbit.walk теперь обычная функция и не имеет [[HomeObject]]. Поэтому в ней не работает super.

Исключением из этого правила являются функции-стрелки. В них используется super внешней функции. Например, здесь функция-стрелка в setTimeout берёт внешний super:

'use strict';

let animal = {

walk() {

console.log("I'm walking");

}

};

let rabbit = {

\_\_proto\_\_: animal,

walk() {

setTimeout(() => super.walk()); // I'm walking

}

};

rabbit.walk();

Ранее мы говорили о том, что у функций-стрелок нет своего this, arguments: они используют те, которые во внешней функции. Теперь к этому списку добавился ещё и super.

Свойство [[HomeObject]] – не изменяемое и поэтому при создании метода – он привязан к своему объекту навсегда. Технически можно даже скопировать его и запустить отдельно, и super продолжит работать:

'use strict';

let animal = {

walk() { console.log("I'm walking"); }

};

let rabbit = {

\_\_proto\_\_: animal,

walk() {

super.walk();

}

};

let walk = rabbit.walk; // скопируем метод в переменную

walk(); // вызовет animal.walk()

В примере выше метод walk() запускается отдельно от объекта, но всё равно, благодаря [[HomeObject]], сохраняется доступ к его прототипу через super.

Это – скорее технический момент, так как методы объекта, всё же, предназначены для вызова в контексте этого объекта. В частности, правила для this в методах – те же, что и для обычных функций. В примере выше при вызове walk() без объекта this будет undefined.

# Классы

В современном JavaScript появился новый, «более красивый» синтаксис для классов. Новая конструкция class – удобный «синтаксический сахар» для задания конструктора вместе с прототипом.

## Class

Синтаксис для классов выглядит так:

class Название [extends Родитель] {

constructor

методы

}

Например:

'use strict';

class User {

constructor(name) {

this.name = name;

}

sayHi() {

console.log(this.name);

}

}

let user = new User("Вася");

user.sayHi(); // Вася

Функция constructor запускается при создании new User, остальные методы записываются в User.prototype.

Это объявление примерно аналогично такому:

function User(name) {

this.name = name;

}

User.prototype.sayHi = function() {

console.log(this.name);

};

В обоих случаях new User будет создавать объекты. Метод sayHi также в обоих случаях находится в прототипе. Но при объявлении через class есть и ряд отличий:

* User нельзя вызывать без new, будет ошибка.
* Объявление класса с точки зрения области видимости ведёт себя как let. В частности, оно видно только в текущем блоке и только в коде, который находится ниже объявления (Function Declaration видно и до объявления).

Методы, объявленные внутри class, также имеют ряд особенностей:

* Метод sayHi является именно методом, то есть имеет доступ к super.
* Все методы класса работают в строгом режиме use strict, даже если он не указан.
* Все методы класса не перечислимы. То есть в цикле for..in по объекту их не будет.

## Class Expression

Также, как и Function Expression, классы можно задавать «инлайн», в любом выражении и внутри вызова функции. Это называется Class Expression

'use strict';

let User = class {

sayHi() { console.log('Привет!'); }

};

new User().sayHi();

В примере выше у класса нет имени, что один-в-один соответствует синтаксису функций. Но имя можно дать. Тогда оно, как и в Named Function Expression, будет доступно только внутри класса:

'use strict';

let SiteGuest = class User {

sayHi() { console.log('Привет!'); }

};

new SiteGuest().sayHi(); // Привет

new User(); // ошибка

В примере выше имя User будет доступно только внутри класса и может быть использовано, например, для создания новых объектов данного типа.

Наиболее очевидная область применения этой возможности – создание вспомогательного класса прямо при вызове функции.

Например, функция createModel в примере ниже создаёт объект по классу и данным, добавляет ему \_id и пишет в «реестр» allModels:

'use strict';

let allModels = {};

function createModel(Model, ...args) {

let model = new Model(...args);

model.\_id = Math.random().toString(36).slice(2);

allModels[model.\_id] = model;

return model;

}

let user = createModel(class User {

constructor(name) {

this.name = name;

}

sayHi() {

console.log(this.name);

}

}, "Вася");

user.sayHi(); // Вася

console.log( allModels[user.\_id].name ); // Вася

## Геттеры, сеттеры и вычисляемые свойства

В классах, как и в обычных объектах, можно объявлять геттеры и сеттеры через get/set, а также использовать […] для свойств с вычисляемыми именами:

'use strict';

class User {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

// геттер

get fullName() {

return `${this.firstName} ${this.lastName}`;

}

// сеттер

set fullName(newValue) {

[this.firstName, this.lastName] = newValue.split(' ');

}

// вычисляемое название метода

["test".toUpperCase()]() {

console.log("PASSED!");

}

};

let user = new User("Вася", "Пупков");

console.log( user.fullName ); // Вася Пупков

user.fullName = "Иван Петров";

console.log( user.fullName ); // Иван Петров

user.TEST(); // PASSED!

При чтении fullName будет вызван метод get fullName(), при присвоении – метод set fullName с новым значением.

Важно заметить, что class не позволяет задавать свойства-значения. В синтаксисе классов, как мы видели выше, можно создавать методы. Они будут записаны в прототип, как например User.prototype.sayHi.

Однако, нет возможности задать в прототипе обычное значение (не функцию), такое как User.prototype.key = "value".

Конечно, никто не мешает после объявления класса в прототип дописать подобные свойства, однако предполагается, что в прототипе должны быть только методы.

Если свойство-значение, всё же, необходимо, то можно создать геттер, который будет нужное значение возвращать.

## Статические свойства

Класс, как и функция, является объектом. Статические свойства класса User – это свойства непосредственно User, то есть доступные из него «через точку».

Для их объявления используется ключевое слово static.

Например:

'use strict';

class User {

constructor(firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

static createGuest() {

return new User("Гость", "Сайта");

}

};

let user = User.createGuest();

console.log( user.firstName ); // Гость

console.log( User.createGuest ); // createGuest ... (функция)

Как правило, они используются для операций, не требующих наличия объекта, например – для фабричных, как в примере выше, то есть как альтернативные варианты конструктора. Или же, можно добавить метод User.compare, который будет сравнивать двух пользователей для целей сортировки.

Также статическими удобно делать константы:

'use strict';

class Menu {

static get elemClass() {

return "menu"

}

}

console.log( Menu.elemClass ); // menu

## Наследование

Синтаксис:

class Child extends Parent {

...

}

Посмотрим, как это выглядит на практике. В примере ниже объявлено два класса: Animal и наследующий от него Rabbit:

'use strict';

class Animal {

constructor(name) {

this.name = name;

}

walk() {

console.log("I walk: " + this.name);

}

}

class Rabbit extends Animal {

walk() {

super.walk();

console.log("...and jump!");

}

}

new Rabbit("Вася").walk();

// I walk: Вася

// and jump!

Как видим, в new Rabbit доступны как свои методы, так и (через super) методы родителя.

Это потому, что при наследовании через extends формируется стандартная цепочка прототипов: методы Rabbit находятся в Rabbit.prototype, методы Animal – в Animal.prototype, и они связаны через \_\_proto\_\_:

'use strict';

class Animal { }

class Rabbit extends Animal { }

console.log( Rabbit.prototype.\_\_proto\_\_ == Animal.prototype ); // true

Как видно из примера выше, методы родителя (walk) можно переопределить в наследнике. При этом для обращения к родительскому методу используют super.walk().

С конструктором – немного особая история. Конструктор constructor родителя наследуется автоматически. То есть, если в потомке не указан свой constructor, то используется родительский. В примере выше Rabbit, таким образом, использует constructor от Animal.

Если же у потомка свой constructor, то, чтобы в нём вызвать конструктор родителя – используется синтаксис super() с аргументами для родителя.

Например, вызовем конструктор Animal в Rabbit:

'use strict';

class Animal {

constructor(name) {

this.name = name;

}

walk() {

console.log("I walk: " + this.name);

}

}

class Rabbit extends Animal {

constructor() {

// вызвать конструктор Animal с аргументом "Кроль"

super("Кроль"); // то же, что и Animal.call(this, "Кроль")

}

}

new Rabbit().walk(); // I walk: Кроль

Для такого вызова есть небольшие ограничения:

* Вызвать конструктор родителя можно только изнутри конструктора потомка. В частности, super() нельзя вызвать из произвольного метода.
* В конструкторе потомка мы обязаны вызвать super() до обращения к this. До вызова super не существует this, так как по спецификации в этом случае именно super инициализирует this.

Второе ограничение выглядит несколько странно, поэтому проиллюстрируем его примером:

'use strict';

class Animal {

constructor(name) {

this.name = name;

}

}

class Rabbit extends Animal {

constructor() {

console.log(this); // ошибка, this не определён!

// обязаны вызвать super() до обращения к this

super();

// а вот здесь уже можно использовать this

}

}

new Rabbit();

# Дополнительные материалы

* <https://learn.javascript.ru/es-object>
* <https://learn.javascript.ru/es-class>

# Задание для закрепления материала

Калькулятор продолжается:

Переписать функционал используя классы