Куда дальше? Новые стандарты

Оглавление

[Синтаксис языка 0](#_Toc444164282)

[Внешние скрипты и порядок их исполнения 1](#_Toc444164283)

[Структура кода 2](#_Toc444164284)

[Комментарии 2](#_Toc444164285)

[Переменные 3](#_Toc444164286)

[Константы 3](#_Toc444164287)

[Типы данных 4](#_Toc444164288)

[Оператор typeof 5](#_Toc444164289)

[Современный стандарт, "use strict" 6](#_Toc444164290)

[Домашнее задание 6](#_Toc444164291)

[Используемая литература 7](#_Toc444164292)

# ES-2015 сейчас

Стандарт ES-2015 был принят в июне 2015. Пока что большинство браузеров реализуют его частично, текущее состояние реализации различных возможностей можно посмотреть здесь: https://kangax.github.io/compat-table/es6/.

Чтобы писать код на ES-2015 прямо сейчас, есть следующие варианты.

**Конкретный движок JS**

Самое простое – это когда нужен один конкретный движок JS, например V8 (Chrome).

Тогда можно использовать только то, что поддерживается именно в нём. Заметим, что в V8 большинство возможностей ES-2015 поддерживаются только при включённом use strict.

При разработке на Node.JS обычно так и делают. Если же нужна кросс-браузерная поддержка, то этот вариант не подойдёт.

**Babel.JS**

Babel.JS – это транспайлер, переписывающий код на ES-2015 в код на предыдущем стандарте ES5.

Он состоит из двух частей:

Собственно транспайлер, который переписывает код.

Полифилл, который добавляет методы Array.from, String.prototype.repeat и другие.

На странице https://babeljs.io/repl/ можно поэкспериментировать с транспайлером: слева вводится код в ES-2015, а справа появляется результат его преобразования в ES5.

Обычно Babel.JS работает на сервере в составе системы сборки JS-кода (например webpack или brunch) и автоматически переписывает весь код в ES5.

Настройка такой конвертации тривиальна, единственно – нужно поднять саму систему сборки, а добавить к ней Babel легко, плагины есть к любой из них.

Если же хочется «поиграться», то можно использовать и браузерный вариант Babel.

Это выглядит так:

<script src="https://js.cx/babel-core/browser.min.js"></script>

<script type="text/babel">

let arr = ["hello", 2]; // let

let [str, times] = arr; // деструктуризация

alert( str.repeat(times) ); // hellohello, метод repeat

</script>

Сверху подключается браузерный скрипт browser.min.js из пакета Babel. Он включает в себя полифилл и транспайлер. Далее он автоматически транслирует и выполняет скрипты с type="text/babel".

Размер browser.min.js превышает 1 мегабайт, поэтому такое использование в production строго не рекомендуется.

Это означает, что при запуске примеров в браузере, который их не поддерживает, будет ошибка. Это не означает, что пример неправильный! Просто пока нет поддержки…

Рекомендуется Chrome Canary, большинство примеров в нём работает. Firefox Developer Edition тоже неплох в поддержке современного стандарта, но на момент написания этого текста переменные let работают только при указании version=1.7 в типе скрипта: <script type="application/javascript;version=1.7">. Надеюсь, скоро это требование (version=1.7) отменят.

Впрочем, если пример в браузере не работает (обычно проявляется как ошибка синтаксиса) – почти все примеры вы можете запустить при помощи Babel, на странице Babel: try it out. Там же увидите и преобразованный код.

На практике для кросс-браузерности всё равно используют Babel.

Ещё раз заметим, что самая актуальная ситуация по поддержке современного стандарта браузерами и транспайлерами отражена на странице <https://kangax.github.io/compat-table/es6/>.

# Переменные let и const

В ES-2015 предусмотрены новые способы объявления переменных: через let и const вместо var.

Например:

let a = 5;

**let**

У объявлений переменной через let есть три основных отличия от var:

**Область видимости переменной let – блок {...}.**

Как мы помним, переменная, объявленная через var, видна везде в функции.

Переменная, объявленная через let, видна только в рамках блока {...}, в котором объявлена.

Это, в частности, влияет на объявления внутри if, while или for.

Например, переменная через var:

var apples = 5;

if (true) {

var apples = 10;

alert(apples); // 10 (внутри блока)

}

alert(apples); // 10 (снаружи блока то же самое)

В примере выше apples – одна переменная на весь код, которая модифицируется в if.

То же самое с let будет работать по-другому:

let apples = 5; // (\*)

if (true) {

let apples = 10;

alert(apples); // 10 (внутри блока)

}

alert(apples); // 5 (снаружи блока значение не изменилось)

Здесь, фактически, две независимые переменные apples, одна – глобальная, вторая – в блоке if.

Заметим, что если объявление let apples в первой строке (\*) удалить, то в последнем alert будет ошибка: переменная неопределена:

if (true) {

let apples = 10;

alert(apples); // 10 (внутри блока)

}

alert(apples); // ошибка!

Это потому что переменная let всегда видна именно в том блоке, где объявлена, и не более.

**Переменная let видна только после объявления.**

Как мы помним, переменные var существуют и до объявления. Они равны undefined:

alert(a); // undefined

var a = 5;

С переменными let всё проще. До объявления их вообще нет.

Такой доступ приведёт к ошибке:

alert(a); // ошибка, нет такой переменной

let a = 5;

Заметим также, что переменные let нельзя повторно объявлять. То есть, такой код выведет ошибку:

let x;

let x; // ошибка: переменная x уже объявлена

Это – хоть и выглядит ограничением по сравнению с var, но на самом деле проблем не создаёт. Например, два таких цикла совсем не конфликтуют:

// каждый цикл имеет свою переменную i

for(let i = 0; i<10; i++) { /\* … \*/ }

for(let i = 0; i<10; i++) { /\* … \*/ }

alert( i ); // ошибка: глобальной i нет

При объявлении внутри цикла переменная i будет видна только в блоке цикла. Она не видна снаружи, поэтому будет ошибка в последнем alert.

**При использовании в цикле, для каждой итерации создаётся своя переменная.**

Переменная var – одна на все итерации цикла и видна даже после цикла:

for(var i=0; i<10; i++) { /\* … \*/ }

alert(i); // 10

С переменной let – всё по-другому.

Каждому повторению цикла соответствует своя независимая переменная let. Если внутри цикла есть вложенные объявления функций, то в замыкании каждой будет та переменная, которая была при соответствующей итерации.

**const**

Объявление const задаёт константу, то есть переменную, которую нельзя менять:

const apple = 5;

apple = 10; // ошибка

В остальном объявление const полностью аналогично let.

Заметим, что если в константу присвоен объект, то от изменения защищена сама константа, но не свойства внутри неё:

const user = {

name: "Вася"

};

user.name = "Петя"; // допустимо

user = 5; // нельзя, будет ошибка

То же самое верно, если константе присвоен массив или другое объектное значение.

# Деструктуризация

Деструктуризация (destructuring assignment) – это особый синтаксис присваивания, при котором можно присвоить массив или объект сразу нескольким переменным, разбив его на части.

**Массив**

Пример деструктуризации массива:

'use strict';

let [firstName, lastName] = ["Вася", "Пупкин"];

alert(firstName); // Вася

alert(lastName); // Пупкин

При таком присвоении первое значение массива пойдёт в переменную firstName, второе – вlastName, а последующие (если есть) – будут отброшены.

Ненужные элементы массива также можно отбросить, поставив лишнюю запятую:

'use strict';

// первый и второй элементы не нужны

let [, , title] = "Юлий Цезарь Император Рима".split(" ");

alert(title); // Император

В коде выше первый и второй элементы массива никуда не записались, они были отброшены. Как, впрочем, и все элементы после третьего.

**Оператор «spread»**

Если мы хотим получить и последующие значения массива, но не уверены в их числе – можно добавить ещё один параметр, который получит «всё остальное», при помощи оператора "..." («spread», троеточие):

'use strict';

let [firstName, lastName, ...rest] = "Юлий Цезарь Император Рима".split(" ");

alert(firstName); // Юлий

alert(lastName); // Цезарь

alert(rest); // Император,Рима (массив из 2х элементов)

Значением rest будет массив из оставшихся элементов массива. Вместо rest можно использовать и другое имя переменной, оператор здесь – троеточие. Оно должно стоять только последним элементом в списке слева.

**Значения по умолчанию**

Если значений в массиве меньше, чем переменных – ошибки не будет, просто присвоится undefined:

'use strict';

let [firstName, lastName] = [];

alert(firstName); // undefined

Впрочем, как правило, в таких случаях задают значение по умолчанию. Для этого нужно после переменной использовать символ = со значением, например:

'use strict';

// значения по умолчанию

let [firstName="Гость", lastName="Анонимный"] = [];

alert(firstName); // Гость

alert(lastName); // Анонимный

В качестве значений по умолчанию можно использовать не только примитивы, но и выражения, даже включающие в себя вызовы функций:

'use strict';

function defaultLastName() {

return Date.now() + '-visitor';

}

// lastName получит значение, соответствующее текущей дате:

let [firstName, lastName=defaultLastName()] = ["Вася"];

alert(firstName); // Вася

alert(lastName); // 1436...-visitor

Заметим, что вызов функции defaultLastName() для генерации значения по умолчанию будет осуществлён только при необходимости, то есть если значения нет в массиве.

**Деструктуризация объекта**

Деструктуризацию можно использовать и с объектами. При этом мы указываем, какие свойства в какие переменные должны «идти».

Базовый синтаксис:

let {var1, var2} = {var1:…, var2…}

Объект справа – уже существующий, который мы хотим разбить на переменные. А слева – список переменных, в которые нужно соответствующие свойства записать.

Например:

'use strict';

let options = {

title: "Меню",

width: 100,

height: 200

};

let {title, width, height} = options;

alert(title); // Меню

alert(width); // 100

alert(height); // 200

Как видно, свойства options.title, options.width и options.height автоматически присвоились соответствующим переменным.

Если хочется присвоить свойство объекта в переменную с другим именем, например, чтобы свойство options.width пошло в переменную w, то можно указать соответствие через двоеточие, вот так:

'use strict';

let options = {

title: "Меню",

width: 100,

height: 200

};

let {width: w, height: h, title} = options;

alert(title); // Меню

alert(w); // 100

alert(h); // 200

В примере выше свойство width отправилось в переменную w, свойство height – в переменную h, аtitle – в переменную с тем же названием.

Если каких-то свойств в объекте нет, можно указать значение по умолчанию через знак равенства =, вот так;

'use strict';

let options = {

title: "Меню"

};

let {width=100, height=200, title} = options;

alert(title); // Меню

alert(width); // 100

alert(height); // 200

Можно и сочетать одновременно двоеточие и равенство:

'use strict';

let options = {

title: "Меню"

};

let {width:w=100, height:h=200, title} = options;

alert(title); // Меню

alert(w); // 100

alert(h); // 200

А что, если в объекте больше значений, чем переменных? Можно ли куда-то присвоить «остаток», аналогично массивам?

Такой возможности в текущем стандарте нет. Она планируется в будущем стандарте, и выглядеть она будет примерно так:

'use strict';

let options = {

title: "Меню",

width: 100,

height: 200

};

let {title, ...size} = options;

// title = "Меню"

// size = { width: 100, height: 200} (остаток)

Этот код будет работать, например, при использовании Babel со включёнными экспериментальными возможностями, но ещё раз заметим, что в текущий стандарт такая возможность не вошла.

**Вложенные деструктуризации**

Если объект или массив содержат другие объекты или массивы, и их тоже хочется разбить на переменные – не проблема.

Деструктуризации можно как угодно сочетать и вкладывать друг в друга.

В коде ниже options содержит подобъект и подмассив. В деструктуризации ниже сохраняется та же структура:

'use strict';

let options = {

size: {

width: 100,

height: 200

},

items: ["Пончик", "Пирожное"]

}

let { title="Меню", size: {width, height}, items: [item1, item2] } = options;

// Меню 100 200 Пончик Пирожное

alert(title); // Меню

alert(width); // 100

alert(height); // 200

alert(item1); // Пончик

alert(item2); // Пирожное

Как видно, весь объект options корректно разбит на переменные.

# Функции

В функциях основные изменения касаются передачи параметров, плюс введена дополнительная короткая запись через стрелочку =>.

**Параметры по умолчанию**

Можно указывать параметры по умолчанию через равенство =, например:

function showMenu(title = "Без заголовка", width = 100, height = 200) {

alert(title + ' ' + width + ' ' + height);

}

showMenu("Меню"); // Меню 100 200

Параметр по умолчанию используется при отсутствующем аргументе или равном undefined, например:

function showMenu(title = "Заголовок", width = 100, height = 200) {

alert('title=' + title + ' width=' + width + ' height=' + height);

}

// По умолчанию будут взяты 1 и 3 параметры

// title=Заголовок width=null height=200

showMenu(undefined, null);

При передаче любого значения, кроме undefined, включая пустую строку, ноль или null, параметр считается переданным, и значение по умолчанию не используется.

**Параметры по умолчанию могут быть не только значениями, но и выражениями.**

Например:

function sayHi(who = getCurrentUser().toUpperCase()) {

alert('Привет, ' + who);

}

function getCurrentUser() {

return 'Вася';

}

sayHi(); // Привет, ВАСЯ

Заметим, что значение выражения getCurrentUser().toUpperCase() будет вычислено, и соответствующие функции вызваны – лишь в том случае, если это необходимо, то есть когда функция вызвана без параметра.

В частности, выражение по умолчанию не вычисляется при объявлении функции. В примере выше функция getCurrentUser() будет вызвана именно в последней строке, так как не передан параметр.

**Оператор spread вместо arguments**

Чтобы получить массив аргументов, можно использовать оператор …, например:

function showName(firstName, lastName, ...rest) {

alert(firstName + ' ' + lastName + ' - ' + rest);

}

// выведет: Юлий Цезарь - Император,Рима

showName("Юлий", "Цезарь", "Император", "Рима");

В rest попадёт массив всех аргументов, начиная со второго.

Заметим, что rest – настоящий массив, с методами map, forEach и другими, в отличие отarguments.

Выше мы увидели использование ... для чтения параметров в объявлении функции. Но этот же оператор можно использовать и при вызове функции, для передачи массива параметров как списка, например:

'use strict';

let numbers = [2, 3, 15];

// Оператор ... в вызове передаст массив как список аргументов

// Этот вызов аналогичен Math.max(2, 3, 15)

let max = Math.max(...numbers);

alert( max ); // 15

Формально говоря, эти два вызова делают одно и то же:

Math.max(...numbers);

Math.max.apply(Math, numbers);

Похоже, что первый – короче и красивее.

**Деструктуризация в параметрах**

Если функция получает объект, то она может его тут же разбить в переменные:

'use strict';

let options = {

title: "Меню",

width: 100,

height: 200

};

function showMenu({title, width, height}) {

alert(title + ' ' + width + ' ' + height); // Меню 100 200

}

showMenu(options);

Можно использовать и более сложную деструктуризацию, с соответствиями и значениями по умолчанию:

'use strict';

let options = {

title: "Меню"

};

function showMenu({title="Заголовок", width:w=100, height:h=200}) {

alert(title + ' ' + w + ' ' + h);

}

// объект options будет разбит на переменные

showMenu(options); // Меню 100 200

Заметим, что в примере выше какой-то аргумент у showMenu() обязательно должен быть, чтобы разбить его на переменные.

Если хочется, чтобы функция могла быть вызвана вообще без аргументов – нужно добавить ей параметр по умолчанию – уже не внутрь деструктуризации, а в самом списке аргументов:

'use strict';

function showMenu({title="Заголовок", width:w=100, height:h=200} = {}) {

alert(title + ' ' + w + ' ' + h);

}

showMenu(); // Заголовок 100 200

В коде выше весь объект аргументов по умолчанию равен пустому объекту {}, поэтому всегда есть что деструктуризовать.

**Имя «name»**

В свойстве name у функции находится её имя.

Например:

'use strict';

function f() {} // f.name == "f"

let g = function g() {}; // g.name == "g"

alert(f.name + ' ' + g.name) // f g

В примере выше показаны Function Declaration и Named Function Expression. В синтаксисе выше довольно очевидно, что у этих функций есть имя name. В конце концов, оно указано в объявлении.

Но современный JavaScript идёт дальше, он старается даже анонимным функциям дать разумные имена.

Например, при создании анонимной функции с одновременной записью в переменную или свойство – её имя равно названию переменной (или свойства).

Например:

'use strict';

// свойство g.name = "g"

let g = function() {};

let user = {

// свойство user.sayHi.name == "sayHi"

sayHi: function() {}

};

**Функции в блоке**

Объявление функции Function Declaration, сделанное в блоке, видно только в этом блоке.

Например:

'use strict';

if (true) {

sayHi(); // работает

function sayHi() {

alert("Привет!");

}

}

sayHi(); // ошибка, функции не существует

То есть, иными словами, такое объявление – ведёт себя в точности как если бы let sayHi = function() {…} было сделано в начале блока.

**Функции через =>**

Появился новый синтаксис для задания функций через «стрелку» =>.

Его простейший вариант выглядит так:

'use strict';

let inc = x => x+1;

alert( inc(1) ); // 2

Эти две записи – примерно аналогичны:

let inc = x => x+1;

let inc = function(x) { return x + 1; };

Как видно, "x => x+1" – это уже готовая функция. Слева от => находится аргумент, а справа – выражение, которое нужно вернуть.

Если аргументов несколько, то нужно обернуть их в скобки, вот так:

'use strict';

let sum = (a,b) => a + b;

// аналог с function

// let inc = function(a, b) { return a + b; };

alert( sum(1, 2) ); // 3

Если нужно задать функцию без аргументов, то также используются скобки, в этом случае – пустые:

'use strict';

// вызов getTime() будет возвращать текущее время

let getTime = () => new Date().getHours() + ':' + new Date().getMinutes();

alert( getTime() ); // текущее время

Когда тело функции достаточно большое, то можно его обернуть в фигурные скобки {…}:

'use strict';

let getTime = () => {

let date = new Date();

let hours = date.getHours();

let minutes = date.getMinutes();

return hourse + ':' + minutes;

};

alert( getTime() ); // текущее время

Заметим, что как только тело функции оборачивается в {…}, то её результат уже не возвращается автоматически. Такая функция должна делать явный return, как в примере выше, если конечно хочет что-либо возвратить.

Функции-стрелки очень удобны в качестве коллбеков, например:

`use strict`;

let arr = [5, 8, 3];

let sorted = arr.sort( (a,b) => a - b );

alert(sorted); // 3, 5, 8

Такая запись – коротка и понятна. Далее мы познакомимся с дополнительными преимуществами использования функций-стрелок для этой цели.

**Функции-стрелки не имеют своего this**

Внутри функций-стрелок – тот же this, что и снаружи.

Это очень удобно в обработчиках событий и коллбэках, например:

'use strict';

let group = {

title: "Наш курс",

students: ["Вася", "Петя", "Даша"],

showList: function() {

this.students.forEach(

student => alert(this.title + ': ' + student)

)

}

}

group.showList();

// Наш курс: Вася

// Наш курс: Петя

// Наш курс: Даша

Здесь в forEach была использована функция-стрелка, поэтому this.title в коллбэке – тот же, что и во внешней функции showList. То есть, в данном случае – group.title.

Если бы в forEach вместо функции-стрелки была обычная функция, то была бы ошибка:

'use strict';

let group = {

title: "Наш курс",

students: ["Вася", "Петя", "Даша"],

showList: function() {

this.students.forEach(function(student) {

alert(this.title + ': ' + student); // будет ошибка

})

}

}

group.showList();

При запуске будет "попытка прочитать свойство title у undefined", так как .forEach(f) при запуске f не ставит this. То есть, this внутри forEach будет undefined.

**Функции-стрелки не имеют своего arguments**

В качестве arguments используются аргументы внешней «обычной» функции.

Например:

'use strict';

function f() {

let showArg = () => alert(arguments[0]);

showArg();

}

f(1); // 1

Вызов showArg() выведет 1, получив его из аргументов функции f. Функция-стрелка здесь вызвана без параметров, но это не важно: arguments всегда берутся из внешней «обычной» функции.

Сохранение внешнего this и arguments удобно использовать для форвардинга вызовов и создания декораторов.

Например, декоратор defer(f, ms) ниже получает функцию f и возвращает обёртку вокруг неё, откладывающую вызов на ms миллисекунд:

'use strict';

function defer(f, ms) {

return function() {

setTimeout(() => f.apply(this, arguments), ms)

}

}

function sayHi(who) {

alert('Привет, ' + who);

}

let sayHiDeferred = defer(sayHi, 2000);

sayHiDeferred("Вася"); // Привет, Вася через 2 секунды

Аналогичная реализация без функции-стрелки выглядела бы так:

function defer(f, ms) {

return function() {

let args = arguments;

let ctx = this;

setTimeout(function() {

return f.apply(ctx, args);

}, ms);

}

}

В этом коде пришлось создавать дополнительные переменные args и ctx для передачи внешних аргументов и контекста через замыкание.

# 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() {

alert(this.name);

}

}

let user = new User("Вася");

user.sayHi(); // Вася

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

Это объявление примерно аналогично такому:

function User(name) {

this.name = name;

}

User.prototype.sayHi = function() {

alert(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.

**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()]() {

alert("PASSED!");

}

};

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

alert( user.fullName ); // Вася Пупков

user.fullName = "Иван Петров";

alert( user.fullName ); // Иван Петров

user.TEST(); // PASSED!

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

**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();

alert( user.firstName ); // Гость

alert( 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"

}

}

alert( 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() {

alert("I walk: " + this.name);

}

}

class Rabbit extends Animal {

walk() {

super.walk();

alert("...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 { }

alert( 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() {

alert("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 инициализует this.

Второе ограничение выглядит несколько странно, поэтому проиллюстрируем его примером:

'use strict';

class Animal {

constructor(name) {

this.name = name;

}

}

class Rabbit extends Animal {

constructor() {

alert(this); // ошибка, this не определён!

// обязаны вызвать super() до обращения к this

super();

// а вот здесь уже можно использовать this

}

}

new Rabbit();

# Домашнее задание

1. Написать задачу gulp для транспиляции кода с помощью babel
2. Потренироваться в использовании нового стандарта, переписывая части предыдущих заданий

# Используемая литература

* [Mozilla Developer Network](https://developer.mozilla.org/)
* [MSDN](http://msdn.microsoft.com/)
* [Safari Developer Library](https://developer.apple.com/library/safari/navigation/index.html)
* [Современный учебник JavaScript](http://learn.javascript.ru)