**Лекция 14.**

**JavaScript: ES6 - let, const, деструктуризация, функции, строки.**

[ECMAScript 3](#_Toc519129716)

[Что такое ES? История версий 3](#_Toc519129717)

[Переменные: let и const 6](#_Toc519129718)

[Let 6](#_Toc519129719)

[Const 9](#_Toc519129720)

[Деструктуризация 10](#_Toc519129721)

[Массив 10](#_Toc519129722)

[Объект 12](#_Toc519129723)

[Вложенные деструктуризации 14](#_Toc519129724)

[Функции 16](#_Toc519129725)

[Параметры по умолчанию 16](#_Toc519129726)

[Оператор spread вместо arguments 16](#_Toc519129727)

[Деструктуризация в параметрах 17](#_Toc519129728)

[Имя «name» 17](#_Toc519129729)

[Функции в блоке 18](#_Toc519129730)

[Функции через => 18](#_Toc519129731)

[Функции-стрелки не имеют своего this 19](#_Toc519129732)

[Функции-стрелки не имеют своего arguments 19](#_Toc519129733)

[Строки 20](#_Toc519129734)

[Строки-шаблоны 20](#_Toc519129735)

[Функции шаблонизации 20](#_Toc519129736)

[Улучшена поддержка юникода 21](#_Toc519129737)

[Полезные методы 22](#_Toc519129738)

[Дополнительные материалы 24](#_Toc519129739)

[Задание для закрепления материала 25](#_Toc519129740)

# ECMAScript

Сначала немного истории. JavaScript создавался как скриптовый язык для Netscape. После чего он был отправлен в ECMA International для стандартизации (ECMA — это ассоциация, деятельность которой посвящена стандартизации информационных и коммуникационных технологий). Это привело к появлению нового языкового стандарта, известного как ECMAScript.

Последующие версии JavaScript уже были основаны на стандарте ECMAScript. Проще говоря, ECMAScript — стандарт, а JavaScript — самая популярная реализация этого стандарта.

## Что такое ES? История версий

ES — это просто сокращение для ECMAScript. Каждое издание ECMAScript получает аббревиатуру ES с последующим его номером. Всего существует 8 версий ECMAScript. ES1 была выпущена в июне 1997 года, ES2 — в июне 1998 года, ES3 — в декабре 1999 года, а версия ES4 — так и не была принята. Не будем углубляться в эти версии, так как они морально устарели, а рассмотрим только последние четыре.

ES5 был выпущен в декабре 2009 года, спустя 10 лет после выхода третьего издания. Среди изменений можно отметить:

* поддержку строгого режима (strict mode);
* аксессоры getters и setters;
* возможность использовать зарезервированные слова в качестве ключей свойств и ставить запятые в конце массива;
* многострочные строковые литералы;
* новую функциональность в стандартной библиотеке;
* поддержку JSON.

Версия ES6/ES2015 вышла в июне 2015 года. Это также принесло некую путаницу в связи с названием пакета, ведь ES6 и ES2015 — это одно и то же. С выходом этого пакета обновлений комитет принял решение перейти к ежегодным обновлениям. Поэтому издание было переименовано в ES2015, чтобы отражать год релиза. Последующие версии также называются в соответствии с годом их выпуска. В этом обновлении были сделаны следующие изменения:

* добавлено деструктурирующее присваивание;
* добавлены стрелочные функции;
* в шаблонных строках можно объявлять строки с помощью ` (обратных кавычек). Шаблонные строки могут быть многострочными, также могут интерполироваться;
* let и const — альтернативы var для объявления переменных. Добавлена «временная мертвая зона»;
* итератор и протокол итерации теперь определяют способ перебора любого объекта, а не только массивов. Symbol используется для присвоения итератора к любому объекту;
* добавлены функции-генераторы. Они используют yield для создания последовательности элементов. Функции-генераторы могут использовать yield\* для делегирования в другую функцию генератора, кроме этого они могут возвращать объект генератора, который реализует оба протокола;
* добавлены промисы.

ES2016 (ES7) вышла в июне 2016 года. Среди изменений в этой версии ECMAScript можно отметить:

* оператор возведения в степень \*\*;
* метод Array.prototype.includes, который проверяет, содержится ли переданный аргумент в массиве.

Спустя еще год выходит версия ES2017 (ES8). Данный стандарт получил следующие изменения:

* асинхронность теперь официально поддерживается (async/await);
* «висячие» запятые в параметрах функций. Добавлена возможность ставить запятые в конце списка аргументов функций;
* добавлено два новых метода для работы со строками: padStart() и padEnd(). Метод padStart() подставляет дополнительные символы слева, перед началом строки. А padEnd(), в свою очередь, справа, после конца строки;
* добавлена функция Object.getOwnPropertyDescriptors(), которая возвращает массив с дескрипторами всех собственных свойств объекта;
* добавлено разделение памяти и объект Atomics.

Что же касается ES.Next, то этот термин является динамическим и автоматически ссылается на новую версию ECMAScript. Стоит отметить, что каждая новая версия приносит новые функции для языка.

# Переменные: let и const

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

## 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.

var apples = 5;

if (true) {

var apples = 10;

console.log(apples); // 10 (внутри блока)

}

console.log(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;

console.log(apples); // 10 (внутри блока)

}

console.log(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 console.log будет ошибка: переменная не определена:

if (true) {

let apples = 10;

console.log(apples); // 10 (внутри блока)

}

console.log(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:

console.log(a); // undefined

var a = 5;

С переменными let всё проще. До объявления их вообще нет. Такой доступ приведёт к ошибке:

console.log(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++) { /\* … \*/ }

console.log( 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 console.log.

**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++) { /\* … \*/ }

console.log(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.

Это позволяет легко решить классическую проблему с замыканиями, описанную в задаче Армия функций.

function makeArmy() {

let shooters = [];

for (let i = 0; i < 10; i++) {

shooters.push(function() {

console.log( i ); // выводит свой номер

});

}

return shooters;

}

var army = makeArmy();

army[0](); // 0

army[5](); // 5

Если бы объявление было var i, то была бы одна переменная i на всю функцию, и вызовы в последних строках выводили бы 10.

А выше объявление let i создаёт для каждого повторения блока в цикле свою переменную, которую функция и получает из замыкания в последних строках.

## 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.

Константы, которые жёстко заданы всегда, во время всей программы, обычно пишутся в верхнем регистре. Например:

const ORANGE = "#ffa500".

Большинство переменных – константы в другом смысле: они не меняются после присвоения. Но при разных запусках функции это значение может быть разным. Для таких переменных можно использовать const и обычные строчные буквы в имени.

# 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] = ["Илья", "Кантор"];

console.log(firstName); // Илья

console.log(lastName); // Кантор

При таком присвоении первое значение массива пойдёт в переменную firstName, второе – в lastName, 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(" ");

console.log(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(" ");

console.log(firstName); // Юлий

console.log(lastName); // Цезарь

console.log(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] = [];

console.log(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"] = [];

console.log(firstName); // Гость

console.log(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"];

console.log(firstName); // Вася

console.log(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.

## Объект

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;

console.log(title); // Меню

console.log(width); // 100

console.log(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;

console.log(title); // Меню

console.log(w); // 100

console.log(h); // 200

В примере выше свойство width отправилось в переменную w, свойство height – в переменную h, а title – 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;

console.log(title); // Меню

console.log(width); // 100

console.log(height); // 200

Можно и сочетать одновременно двоеточие и равенство:

'use strict';

let options = {

title: "Меню"

};

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

console.log(title); // Меню

console.log(w); // 100

console.log(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 с включёнными экспериментальными возможностями, но ещё раз заметим, что в текущий стандарт такая возможность не вошла.

В примерах выше переменные объявлялись прямо перед присваиванием: let {…} = {…}. Конечно, можно и без let, использовать уже существующие переменные.

Однако, здесь есть небольшой «подвох». В JavaScript, если в основном потоке кода (не внутри другого выражения) встречается {...}, то это воспринимается как блок.

Например, можно использовать такой блок для ограничения видимости переменных:

'use strict';

{

// вспомогательные переменные, локальные для блока

let a = 5;

// поработали с ними

console.log(a); // 5

// больше эти переменные не нужны

}

console.log(a); // ошибка нет такой переменной

Конечно, это бывает удобно, но в данном случае это создаст проблему при деструктуризации:

let a, b;

{a, b} = {a:5, b:6}; // будет ошибка, оно посчитает, что {a,b} - блок

Чтобы избежать интерпретации {a, b} как блока, нужно обернуть всё присваивание в скобки:

let a, b;

({a, b} = {a:5, b:6}); // внутри выражения это уже не блок

## 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

console.log(title); // Меню

console.log(width); // 100

console.log(height); // 200

console.log(item1); // Пончик

console.log(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, 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 (то есть функция), и стоит заметить, что эти выражения будут вызывать только если этот параметр не был передан или равен undefined.

## 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 с третьего. Заметим, что rest – настоящий массив, с методами map, forEach и другими, в отличие от arguments. А так же важно помнить, что этот оператор должен быть обязательно в конце списка параметров.

Так же spread можно использовать и при вызове функции:

'use strict';

let numbers = [2, 3, 15];

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

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

alert( max ); // 15

## 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);

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

## Имя «name»

В свойстве name у функции находится её имя. Например:

'use strict';

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

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

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

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

## Функции через =>

Появился новый синтаксис для задания функций через «стрелку» =>. Его простейший вариант выглядит так:

'use strict';

let inc = x => x+1;

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

alert( inc(1) ); // 2

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:

'use strict';

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

// аналог let sum = 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. Когда тело функции достаточно большое, то можно его обернуть в фигурные скобки {…}. Заметим, что как только тело функции оборачивается в {…}, то, её результат уже не возвращается автоматически. Такая функция должна делать явный return

## Функции-стрелки не имеют своего this

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

Функции стрелки нельзя запускать с new. Отсутствие у функции-стрелки "своего this" влечёт за собой естественное ограничение: такие функции нельзя использовать в качестве конструктора, то есть нельзя вызывать через new.

Есть тонкое различие между функцией стрелкой => и обычной функцией, у которой вызван .bind(this):

* Вызовом .bind(this) мы передаём текущий this, привязывая его к функции.
* При => привязки не происходит, так как функция стрелка вообще не имеет контекста this. Поиск this в ней осуществляется так же, как и поиск обычной переменной, то есть, выше в замыкании. До появления стандарта ES-2015 такое было невозможно.

## Функции-стрелки не имеют своего arguments

В качестве arguments используются аргументы внешней «обычной» функции. Сохранение внешнего this и arguments удобно использовать для форвардинга вызовов и создания декораторов (не приходится создавать дополнительные переменные для передачи внешних аргументов и контекста через замыкание).

# Строки

Есть ряд улучшений и новых методов для строк. Начнём с, пожалуй, самого важного.

## Строки-шаблоны

Добавлен новый вид кавычек для строк:

let str = `обратные кавычки`;

Основные отличия от двойных "…" и одинарных '…' кавычек:

1. В них разрешён перевод строки.

alert(`моя

многострочная

строка`);

Заметим, что пробелы и, собственно, перевод строки также входят в строку, и будут выведены.

1. Можно вставлять выражения при помощи ${…}.

'use strict';

let apples = 2;

let oranges = 3;

alert(`${apples} + ${oranges} = ${apples + oranges}`); // 2 + 3 = 5

Как видно, при помощи ${…} можно вставлять как и значение переменной ${apples}, так и более сложные выражения, которые могут включать в себя операторы, вызовы функций и т.п. Такую вставку называют «интерполяцией».

## Функции шаблонизации

Можно использовать свою функцию шаблонизации для строк. Название этой функции ставится перед первой обратной кавычкой. Такая функция будет автоматически вызвана и получит в качестве аргументов строку, разбитую по вхождениям параметров ${…} и сами эти параметры.

Например:

'use strict';

function f(strings, ...values) {

alert(JSON.stringify(strings)); // ["Sum of "," + "," =\n ","!"]

alert(JSON.stringify(strings.raw)); // ["Sum of "," + "," =\\n ","!"]

alert(JSON.stringify(values)); // [3,5,8]

}

let apples = 3;

let oranges = 5;

let str = f`Sum of ${apples} + ${oranges} =\n ${apples + oranges}!`;

В примере выше видно, что строка разбивается по очереди на части: «кусок строки» – «параметр» – «кусок строки» – «параметр».

* Участки строки идут в первый аргумент-массив strings.
* У этого массива есть дополнительное свойство strings.raw. В нём находятся строки в точности как в оригинале. Это влияет на спец-символы, например в strings символ \n – это перевод строки, а в strings.raw – это именно два символа \n.
* Дальнейший список аргументов функции шаблонизации – это значения выражений в ${...}, в данном случае их три.

Функция шаблонизации – это своего рода «стандартный синтаксический сахар» для упрощения форматирования и парсинга строк.

## Улучшена поддержка юникода

Внутренняя кодировка строк в JavaScript – это UTF-16, то есть под каждый символ отводится ровно два байта.

Но под всевозможные символы всех языков мира 2 байт не хватает. Поэтому бывает так, что одному символу языка соответствует два юникодных символа (итого 4 байта). Такое сочетание называют «суррогатной парой».

Самый частый пример суррогатной пары, который можно встретить в литературе – это китайские иероглифы или редкие математические символы, а также некоторые символы для эмоций и так далее.

В современный JavaScript добавлены методы String.fromCodePoint и str.codePointAt – аналоги String.fromCharCode и str.charCodeAt, корректно работающие с суррогатными парами.

Например, charCodeAt считает суррогатную пару двумя разными символами и возвращает код каждой, в то время как codePointAt возвращает его Unicode-код суррогатной пары правильно:

alert( '𝒳'.charCodeAt(0) + ' ' + '𝒳'.charCodeAt(1) ); // 55349 56499

alert( '𝒳'.codePointAt(0) ); // 119987

Метод String.fromCodePoint(code) корректно создаёт строку из «длинного кода», в отличие от старого String.fromCharCode(code).

// Правильно

alert( String.fromCodePoint(119987) ); // 𝒳

// Неверно!

alert( String.fromCharCode(119987) ); // 풳

**\u{длинный код}**

Есть и ещё синтаксическое улучшение для больших Unicode-кодов. В JavaScript-строках давно можно вставлять символы по Unicode-коду, вот так:

alert( "\u2033" ); // ″, символ двойного штриха

Синтаксис: \uNNNN, где NNNN – четырёхзначный шестнадцатиричный код, причём он должен быть ровно четырёхзначным. «Лишние» цифры уже не войдут в код.

Чтобы вводить более длинные коды символов, добавили запись \u{NNNNNNNN}, где NNNNNNNN – максимально восьмизначный (но можно и меньше цифр) код.

## Полезные методы

Добавлен ряд полезных методов общего назначения:

* str.includes(s) – проверяет, включает ли одна строка в себя другую, возвращает true/false.
* str.endsWith(s) – возвращает true, если строка str заканчивается подстрокой s.
* str.startsWith(s) – возвращает true, если строка str начинается со строки s.
* str.repeat(times) – повторяет строку str times раз.

Конечно, всё это можно было сделать при помощи других встроенных методов, но новые методы более удобны.

# Дополнительные материалы

* <https://learn.javascript.ru/es-modern-usage>
* <https://learn.javascript.ru/let-const>
* <https://learn.javascript.ru/destructuring>
* <https://learn.javascript.ru/es-function>
* <https://learn.javascript.ru/es-string>
* <https://tproger.ru/translations/wtf-is-ecmascript/>
* <https://tproger.ru/translations/es6-cheatsheet-1/>
* <https://tproger.ru/translations/es6-cheatsheet-2/>

# Задание для закрепления материала

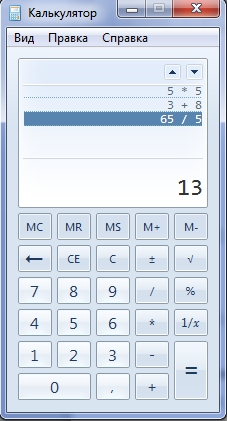
Калькулятор продолжается:

**Приоритет (не переходить к следующей секции не выполнив это):**

1. Теперь сборки должно быть две: dist/prod и dist/dev
   1. dist/prod
      1. запускается с помощью команды «npm run build:prod»
      2. все файлы минифицируются
      3. в index.html должно выводиться «Production mode» в заголовке (делается через webpack, плагин HtmlWebpackPlugin)
   2. dist/dev
      1. запускается с помощью команды «npm run build:dev»
      2. все файлы собираются в один, но не минифицируются
      3. в index.html должно выводиться «Development mode» в заголовке
2. Переписать существующий код под возможности ES6 (которые есть в лекции) и указать в отдельной секции ReadMe какие возможности были использованы и зачем.
3. Подключить транспайлер BabelJS к сборке и компилировать JS через него (только через webpack)

**Дополнительно (не означает «не обязательно»):**

1. Добавить в калькулятор унарный минус
2. Добавить опциональный журнал вычислений (не путать с историей). Опциональный означает, что я его могу включить/выключить (показать/скрыть). Журнал вычислений означает, что я могу посмотреть, какие вычисления я делал в текущий сеанс работы. С возможностью его очистки. Пример такой функциональности есть в калькуляторе Windows7 (вид -> журнал). Каких либо дополнительных возможностей не требуется. Прокрутка журнала может осуществляться стандартным скроллом элемента textarea. Скриншот прилагается далее



P.S. Нарушение приоритета будет считаться минусом при оценки работы