



# РАСШИРЕННЫЙ СИНТАКСИС В JAVASCRIPT



ВЛАДИМИР ЧЕБУКИН



# ВЛАДИМИР ЧЕБУКИН

Frontend-разработчик



[vovachebr@mail.ru](mailto:vovachebr@mail.ru)



[fb.me/vovachebr](https://fb.me/vovachebr)



[@User123423](https://t.me/User123423)



# ПЛАН ЗАНЯТИЯ

1. Современный стандарт, "use strict"
2. Операторы разделители
3. Типы null и undefined
4. Управляющие конструкции
5. Массивы, циклы

# ВСПОМНИМ ПРОШЛЫЙ МАТЕРИАЛ

Что будет выведено?

```
1 console.log("abc" + "abc");  
2 console.log("abc" + 3);  
3 console.log(`${4+6} = 15`);
```

# ВСПОМНИМ ПРОШЛЫЙ МАТЕРИАЛ

Вывод будет таким:

```
1 console.log("abc" + "abc"); // "abcabc"  
2 console.log("abc" + 3); // "abc3"  
3 console.log(`${4+6} = 15`); // 10 = 15
```

# СОВРЕМЕННЫЕ СТАНДАРТЫ

20 лет назад, когда язык JavaScript только придумали не было никаких правил его написания. Каждый браузер обрабатывал JavaScript-код по-своему. Разработчикам приходилось придумывать механизмы работы сайтов для разных браузеров.

## ПРИМЕР

Рассмотрим большую группу людей, говорящих по-английски. Кто-то среди них может знать определенные слова, выражения и синтаксические правила, не известные другим людям, и наоборот. То же самое с браузерами. Хотя все браузерные JavaScript-движки понимают JavaScript, некоторые понимают его лучше других. Различаются и способы поддержки языка браузерами.



# СОВРЕМЕННЫЕ СТАНДАРТЫ

В 1997 году придумали первые правила, они назывались спецификациями. Создатели сайтов постепенно внедряли стандарты в браузеры.

Если взять код, который был написан в 1995 году, он не воспроизведется сегодня на компьютере, т.к. браузеры с тех пор сильно изменились.



# СОВРЕМЕННЫЕ СТАНДАРТЫ ЯЗЫКА JAVASCRIPT


Спецификация ECMAScript 5 (была выпущена в декабре 2009 года).  
Спецификация ECMAScript 6 (ES2015) (была выпущен в 2015 году) —  
Является более актуальной версией языка.

Более современные версии языка:

- ES2016 (ES7)
- ES2017 (ES8)
- ES2018 (ES9)

[Обзор ECMAScript 2016, 2017, и 2018](#)





# КАКУЮ СПЕЦИФИКАЦИЮ БУДЕМ ИСПОЛЬЗОВАТЬ МЫ?

В нашем курсе будем использовать ES6, т.к. ей пользуется большинство разработчиков, а также присутствует поддержка большинства современных браузеров



# ИСПОЛЬЗОВАНИЕ СОВРЕМЕННЫХ ВОЗМОЖНОСТЕЙ ЯЗЫКА

Чтобы решить эту проблему, существует руководящее указание

```
"use strict";
```

# ЗАЧЕМ ИСПОЛЬЗОВАТЬ "USE STRICT";?

`"use strict"` — это директива, которая заставляет код обрабатываться в строгом режиме (способ обеспечения более тщательной проверки ошибок в коде). Без этой директивы код обрабатывается в неограниченном режиме.

В строгом режиме нельзя, например, использовать неявно объявляемые переменные, присваивать значения свойствам, доступным только для чтения, и добавлять свойства в объекты, которые не являются расширяемыми.

1. некоторые ошибки можно найти быстрее;
2. более опасные и бесполезные черты JavaScript'a либо запрещены, либо приводят к ошибке.

# КАК ИСПОЛЬЗОВАТЬ "USE STRICT";?

Чтобы включить строгий режим в целом скрипте, надо поставить директиву "use strict"; или 'use strict'; в начало скрипта.

```
"use strict";  
// код здесь обрабатывается в строгом режиме
```

Чтобы включить строгий режим в функции, надо поставить директиву в начало кода функции.

```
1 // код здесь обрабатывается в неограниченном режиме  
2 function f() {  
3     "use strict";  
4     // код здесь обрабатывается в строгом режиме  
5 }  
6 // код здесь обрабатывается в неограниченном режиме
```

# ОТМЕНИТЬ ДЕЙСТВИЕ USE STRICT НИКАК НЕЛЬЗЯ

Не существует директивы `no use strict` или подобной, которая возвращает в старый режим.

Если уж вошли в современный режим, то это дорога в один конец.  
В рамках нашего курса использование "use strict" обязательное условие, а его отсутствие считается плохим тоном.



# ИЗМЕНЕНИЯ В СТРОГОМ РЕЖИМЕ

1. Преобразование ошибок в исключения;
2. Упрощение работы с переменными;
3. Упрощение eval и arguments;
4. «Обезопасенный» JavaScript;
5. Подготовка почвы для будущих версий ECMAScript.

[Зачем в JavaScript нужен строгий режим?](#)



# ОПЕРАТОРЫ РАЗДЕЛИТЕЛИ

Используются, когда необходимо разделить различные вычисления.



# ВСПОМИНАЕМ ПРОШЛЫЕ ЗАНЯТИЕ

Какие существуют операторы?

На этом занятии познакомимся с операторами *запятой* и *точки с запятой*.



# ОПЕРАТОР ЗАПЯТАЯ

Оператор запятая выполняет каждый из его операндов (слева направо) и возвращает значение последнего операнда.

```
1  let a = (7, 5);  
2  a; // 5  
3  
4  let x, y, z  
5  x = (y=1, z=4);  
6  x; // 4  
7  y; // 1  
8  z; // 4
```



# АКТУАЛЬНОСТЬ СКОБОК

Приоритет операторов определяет, в каком порядке будут выполнены операнды внутри выражения.

- Скобки имеют наивысший приоритет;
- Оператор запятая имеет наименьший приоритет из всех операторов.

## ПРИМЕР АКТУАЛЬНОСТИ СКОБОК

```
1 console.log(1 + 2 * 3); // 1 + 6
2 console.log(1 + (2 * 3)); // 1 + 6
3 console.log((1 + 2) * 3); // 3 * 3
4 console.log(1 * 3 + 2 * 3); // 3 + 6
```

# ЗАПЯТЫЕ В ЗНАЧЕНИЯХ ТИПА И В ОБЪЯВЛЕНИЯХ

```
1 // создает массив из 4 элементов
2 let arr = [1, 2, 3, 4];
3
4 //определяет три переменные
5 let a = 1, b = 2, c = 3;
6
7 //вызывает функцию, передавая 2 параметра
8 Math.max(4, 7);
```



# ОПЕРАТОР ТОЧКА С ЗАПЯТОЙ

Точка с запятой — это разделитель объявлений, а запятая — это разделитель выражений внутри объявлений.

Точки с запятой нужно ставить, даже если их, казалось бы, можно пропустить.

Есть языки, в которых точка с запятой необязательна, и её там никто не ставит. В JavaScript перевод строки её заменяет, но лишь частично, поэтому лучше её ставить.



## ПУСТОЕ ЗНАЧЕНИЕ (ОТСУТСТВИЕ ЗНАЧЕНИЯ)

Представьте, что вам необходимо показать отсутствие какого-либо значения.

Пример: вы создаете список друзей и их номера телефонов. В некоторых случаях вы можете не знать номер телефона друга (**undefined**), а в некоторых случаях у друга вообще нет телефона (**null**).



## ТИП UNDEFINED

`undefined` является свойством глобального объекта, то есть это переменная в глобальной области видимости. Начальным значением `undefined` является примитивное значение `undefined`.

# ТИП NULL

```
let phoneNumber = null;
```

Значение `null` является литералом JavaScript, представляющим нулевое или «пустое» значение, то есть, когда нет никакого объектного значения. Это одно из примитивных значений JavaScript.

В частности, код выше говорит о том, что номер телефона `phoneNumber` неизвестен.



# ОПИСАНИЕ NULL

Значение `null` является значением (а не свойством глобального объекта, как `undefined`). В программах `null` часто присутствует в местах, где ожидается объект, но подходящего объекта нет. При проверке на `null` или `undefined` помните о различии между операторами равенства `==` и идентичности `===` (с первым выполняется преобразование типов).

```
1 // переменная foo не существует, она не была определена
2 // и никогда не инициализировалась:
3 > foo
4 'ReferenceError: foo is not defined'
5
6 // переменная foo сейчас существует, но она не имеет ни типа, ни значения:
7 > let foo = null; foo
8 'null'
```



# СРАВНЕНИЕ ЗНАЧЕНИЙ

Пример: необходимо сравнить товары в магазине и выбрать набор покупок.

# ОПЕРАТОРЫ СРАВНЕНИЯ И ЛОГИЧЕСКИЕ ЗНАЧЕНИЯ

Многие операторы сравнения знакомы нам из математики:

- Больше/меньше:  $a > b$ ,  $a < b$ ;
- Больше/меньше или равно:  $a \geq b$ ,  $a \leq b$ ;
- Равно  $a == b$ . Для сравнения используется два символа равенства '='. Один символ  $a = b$  означал бы присваивание.
- «Не равно». В математике он пишется как  $\neq$ , в JavaScript — знак равенства с восклицательным знаком перед ним  $!=$ .

# СРАВНЕНИЕ СТРОК

Строки сравниваются побуквенно:

```
'Б' > 'А' // true
```

Аналогом «алфавита» во внутреннем представлении строк служит кодировка, у каждого символа — свой номер (код). JavaScript использует кодировку Unicode.

# СРАВНЕНИЕ РАЗНЫХ ТИПОВ

При сравнении значений разных типов, используется числовое преобразование. Оно применяется к обоим значениям.

Например:

```
1 console.log( '4' > 1 ); // true, сравнивается как 4 > 1
2 console.log( '02' == 2 ); // true, сравнивается как 2 == 2
3 console.log( true == 1 ); // true, так как true становится числом 1
4 console.log( false == 0 ); // true, false становится числом 0.
```

# СТРОГОЕ РАВЕНСТВО

В обычном операторе `==` есть «проблема» — в случае сравнения значений разных типов одно значение преобразуется к типу другого значения, в некоторых случаях этот оператор может сыграть злую шутку.

```
console.log( 0 == false ); // true  
console.log( '' == false ); // true
```

Для проверки равенства без преобразования типов используются операторы строгого равенства `===` (тройное равно) и `!==`.

```
console.log( 0 === false ); // false, т.к. типы различны
```



# ЛОГИЧЕСКИЕ ОПЕРАТОРЫ

Для операций над логическими значениями в JavaScript есть `||` (ИЛИ), `&&` (И) и `!` (НЕ).

Хоть они и называются «логическими», но в JavaScript могут применяться к значениям любого типа и возвращают также значения любого типа.

## ОПЕРАТОР || (ИЛИ)

Оператор ИЛИ выглядит как двойной символ вертикальной черты:

```
result = a || b;
```

Логическое ИЛИ в классическом программировании работает следующим образом: "если хотя бы один из аргументов true, то возвращает true, иначе – false".

```
console.log( true || true ); // true
console.log( false || true ); // true
console.log( true || false ); // true
console.log( false || false ); // false
```



## ПРИМЕР ПОКУПОК В МАГАЗИНЕ

```
1 let milk = 60;  
2 let bread = 30;  
3 let fruits = 80;  
4 console.log(milk+bread < 70 || fruits <= 100)
```

# КОРОТКИЙ ЦИКЛ ВЫЧИСЛЕНИЙ

JavaScript вычисляет несколько ИЛИ слева направо. При этом, чтобы сэкономить ресурсы, используется так называемый «короткий цикл вычисления».

Допустим, вычисляются несколько ИЛИ подряд: `a || b || c || ...`. Если первый аргумент — `true`, то результат заведомо будет `true` (хотя бы одно из значений — `true`), и остальные значения игнорируются.

```
1  let x;  
2  true || (x = 1);  
3  console.log(x); // undefined, x не присвоен  
4  false || (x = 1);  
5  console.log(x); // 1
```

Если все значения «ложные», то `||` возвратит последнее из них:

# ОПЕРАТОР && (И)

Оператор И выглядит как амперсанда &&:

```
result = a && b;
```

Логическое И в классическом программировании работает следующим образом: «если хотя бы один из аргументов false, то возвращает false, иначе – true».

```
console.log( true && true ); // true
console.log( false && true ); // false
console.log( true && false ); // false
console.log( false && false ); // false
```

К И применим тот же принцип «короткого цикла вычислений», но немного по-другому, чем к ИЛИ.

Если левый аргумент – false, оператор И возвращает его и заканчивает вычисления. Иначе – вычисляет и возвращает правый аргумент.

# ОПЕРАТОР !(НЕ)

Оператор НЕ — самый простой. Он получает один аргумент. Синтаксис:

```
let result = !value;
```

1. Сначала приводит аргумент к логическому типу true/false.
2. Затем возвращает противоположное значение.

```
console.log( !true ); // false  
console.log( !0 ); // true
```

**В частности, двойное НЕ используют для преобразования значений к логическому типу**

```
console.log( !!{name:"Vasia", age:36} ); // true
```



## ВЫБОР ДЕЙСТВИЯ

Пример: на перекрестке можно повернуть направо, налево или пойти прямо.

Пример из программирования: если пользователь ввел верный логин/пароль, то авторизовать его на сайте.

# УСЛОВНЫЕ КОНСТРУКЦИИ

Иногда, в зависимости от условия, нужно выполнить различные действия. Для этого используется оператор `if`.

```
1 let year = new Date().getFullYear();  
2 if (year !== 2018){  
3     console.log("Эта презентация была сделана в этом году");  
4 }
```

# ОПЕРАТОР IF

Оператор if («если») получает условие. Он вычисляет его, и если результат — true, то выполняет команду.

- Число 0, пустая строка "", null и undefined, а также NaN являются false
- Остальные значения — true.

## БЛОК ELSE

Необязательный блок else («иначе») выполняется, если условие неверно:

```
1 let year = new Date().getFullYear();
2 if (year !== 2018){
3     console.log("Эта презентация была сделана в этом году");
4 }else{
5     console.log("Эта презентация была сделана в 2018 году");
6 }
```



# НЕСКОЛЬКО УСЛОВИЙ, ELSE IF

Бывает нужно проверить несколько вариантов условия.  
Для этого используется блок else if .... Например:

```
1  let hours = new Date().getHours();
2  if (hours >= 6 && hours < 12){
3      console.log("Доброе утро");
4  }
5  else if (hours < 18){
6      console.log("Добрый день");
7  }
8  else if (hours < 22){
9      console.log("Добрый вечер");
10 }
11 else{
12     console.log("Доброй ночи");
13 }
```

# ОТСУТСТВИЕ ФИГУРНЫХ СКОБОК (ДЕМО)

Если в условном операторе присутствует только одно действие, то фигурные скобки ставить не обязательно

```
1 let hours = new Date().getHours();
2 if (hours >= 6 && hours < 12)
3     console.log("Доброе утро");
4 else if (hours < 18)
5     console.log("Добрый день");
6 else if (hours < 22)
7     console.log("Добрый вечер");
8 else
9     console.log("Доброй ночи");
```



# КОНСТРУКЦИЯ SWITCH

Конструкция switch заменяет собой сразу несколько if.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

# СИНТАКСИС SWITCH

```
1  switch(x) {  
2      case 'value1': // if (x === 'value1')  
3          ...  
4          [break]  
5  
6      case 'value2': // if (x === 'value2')  
7          ...  
8          [break]  
9  
10     default:  
11         ...  
12         [break]  
13 }
```

## СИНТАКСИС SWITCH

- Переменная `x` проверяется на строгое равенство первому значению `value1`, затем второму `value2` и так далее.
- Если соответствие установлено — `switch` начинает выполняться от соответствующей директивы `case` и далее, до ближайшего `break` (или до конца `switch`).
- Если ни один `case` не совпал — выполняется (если есть) вариант `default`.

# ГРУППИРОВКА CASE (ДЕМО)

Несколько значений case можно группировать.

В примере ниже case 3 и case 5 выполняют один и тот же код:

```
1  let a = 5+7;
2
3  switch (a) {
4      case 12:
5          alert('Верно!');
6          break;
7
8      case 11:
9      case 13:
10         alert('Неверно!');
11         alert('Немного ошиблись, бывает. ');
12         break;
13
14     default:
15         alert('Странный результат, очень странный');
16 }
```

# ТЕРНАРНЫЙ ОПЕРАТОР

Иногда нужно в зависимости от условия присвоить переменную.

условие ? значение1 : значение2

Проверяется условие, затем если оно верно — возвращается значение1, если неверно — значение2, например:

```
let age = 18
access = (age > 14) ? true : false;
greeting = (age >= 18) ? "Здравствуйте" : "Привет";
```

*В данном случае можно было бы обойтись и без оператора '?',*

*т.к. сравнение само по себе уже возвращает true/false:*

Вопросительный знак — единственный оператор, у которого есть аж три аргумента, в то время как у обычных операторов их один-два.

Поэтому его называют «тернарный оператор»



# МАССИВЫ

Пример: необходимо хранить список телефонных номеров друзей.





# МАССИВЫ

Массив — разновидность объекта, которая предназначена для хранения пронумерованных значений и предлагает дополнительные методы для удобного манипулирования такой коллекцией.

# ЭЛЕМЕНТЫ МАССИВОВ

Элементы нумеруются, начиная с нуля.

Чтобы получить нужный элемент из массива — указывается его номер в квадратных скобках.

Массив `buildings` с тремя элементами:

```
1 let buildings = ["Pool", "Shop", "Market"];
2 buildings[3] = "House";
3 buildings[5] = "Hospital";
4 // Попробуйте догадаться какой будет вывод?
5 console.log(buildings[0]);
6 console.log(buildings[3]);
7 console.log(buildings);
```

В массиве может храниться любое число элементов любого типа.

# ПОЯСНЕНИЕ

```
1 let buildings = ["Pool", "Shop", "Market"];
2 // Сейчас в массиве элементы: [0]="Pool" [1]="Shop" [2]="Market"
3 buildings[3] = "House";
4 /* Теперь в массиве элементы:
5 [0]="Pool" [1]="Shop" [2]="Market" [3]="House"*/
6 buildings[5] = "Hospital";
7 /* Теперь в массиве элементы:
8 [0]="Pool" [1]="Shop" [2]="Market" [3]="House" [4]=empty [5]="Hospital"*/
```

# МЕТОДЫ ДОБАВЛЕНИЯ/УДАЛЕНИЯ ЭЛЕМЕНТОВ ИЗ МАССИВА

Использование массива в качестве стека (работает по принципу последний вошел первый вышел)

1. Функция `pop()` удаляет последний элемент из массива и возвращает его
2. Функция `push()` добавляет элемент в конец массива и возвращает количество элементов

Использование массива в качестве очереди (работает по принципу первый вошел первый вышел)

1. Функция `shift()` удаляет из массива первый элемент и возвращает его
2. Функция `unshift()` добавляет элемент в начало массива

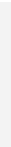
## ПРИМЕРЫ (ДЕМО)

```
1 let buildings = ["Pool", "Shop", "Market"];
2 console.log(buildings.pop()); // "Market"
3 console.log(buildings); // "Pool", "Shop"
4 console.log(buildings.push("Hospital")); // 3
5 console.log(buildings); // "Pool", "Shop", "Hospital"
6 console.log(buildings.shift()); // "Pool"
7 console.log(buildings); // "Shop", "Hospital"
8 console.log(buildings.unshift("Market")); // 3
9 console.log(buildings); // "Market", "Shop", "Hospital"
```



# ЦИКЛЫ

Пример: попробуйте вывести числа от 1 до 1000.



При использовании цикла нет необходимости писать огромное количество строк кода.



# ЦИКЛЫ

Циклы используются для перебора элементов в коллекции, а также для многократного повторения одного участка кода.

# ЦИКЛ FOR

Внимание! Эта информация будет полезна вам при решении ДЗ!

Чаще всего применяется цикл for. Выглядит он так:

```
1  for (начало; условие; шаг) {  
2      // ... тело цикла ...  
3  }
```

```
1  for (let i = 0; i <= 1000; i++) {  
2      console.log(i);  
3  }
```



# ПРОПУСК ЧАСТЕЙ FOR

Любая часть for может быть пропущена.

На примере ниже пропущены все части, таким образом получится бесконечный цикл

```
1 | for (;;) {  
2 |     // будет выполняться вечно  
3 | }
```

## ЦИКЛ FOR..OF

Оператор `for...of` выполняет цикл обхода итерируемых объектов, вызывая на каждом шаге итерации операторы для каждого значения из различных свойств объекта.

```
1  |  for (переменная of объект){  
2  |      оператор  
3  |  }
```

```
1  |  let buildings = ["Pool", "Shop", "Market"];  
2  |  for (building of buildings){  
3  |      console.log(building)  
4  |  }
```

# ЦИКЛ WHILE

Цикл while имеет вид:

```
1  while (условие) {  
2      // код, тело цикла  
3  }
```

```
1  let i = 0;  
2  while (i<100) {  
3      console.log(i);  
4      i++;  
5  }  
6  // будут выведены числа от 0 до 99
```

Пока условие верно — выполняется код из тела цикла. Цикл do...while

# ЦИКЛ DO...WHILE

Проверку условия можно поставить под телом цикла, используя специальный синтаксис do..while:

```
1  do {  
2    // тело цикла  
3  } while (условие);
```

```
1  let i = 0;  
2  do {  
3    console.log(i); // 0  
4    i++; // i=1  
5  } while (i > 100) //false, цикл прекратиться
```

Цикл, описанный, таким образом, сначала выполняет тело, а затем проверяет условие.

## ПРЕРЫВАНИЕ ЦИКЛА: BREAK

Выйти из цикла можно не только при проверке условия но и, вообще, в любой момент. Эту возможность обеспечивает директива `break`.

*Вообще, сочетание «бесконечный цикл + `break`» — отличная штука для тех ситуаций, когда условие, по которому нужно прерваться, находится не в начале-конце цикла, а посередине.*

## СЛЕДУЮЩАЯ ИТЕРАЦИЯ CONTINUE (ДЕМО BREAK + CONTINUE)

Директива `continue` прекращает выполнение текущей итерации цикла.

Она — в некотором роде «младшая сестра» директивы `break`: прерывает не весь цикл, а только текущее выполнение его тела, как будто оно закончилось.

Её используют, если понятно, что на текущем повторе цикла делать больше нечего.

# ЧЕМУ МЫ НАУЧИЛИСЬ?

1. Узнали спецификации языка JavaScript, и назначение директивы "use strict";
2. Операторы запятой и точки с запятой, когда и зачем их использовать;
3. Типы null и undefined, их назначение и особенности;
4. Выполнение кода в зависимости от различных условий при помощи условных конструкций;
5. Массивы, возможности итерирования по массивам и различные циклы, для вычисления подобных действий.



# ДОМАШНЕЕ ЗАДАНИЕ

Давайте посмотрим ваше [домашнее задание](#).

- Вопросы по домашней работе задаем в чате Slack!
- Задачи можно сдавать по частям.
- Зачет по домашней работе проставляется после того, как приняты все задачи.





Спасибо за внимание!  
Время задавать вопросы 😊

**ВЛАДИМИР ЧЕБУКИН**

 [vovachebr@mail.ru](mailto:vovachebr@mail.ru)

 [fb.me/vovachebr](https://fb.me/vovachebr)

 [@User123423](https://t.me/@User123423)