Базовые операторы, математика

**Операнд** – то, к чему применяется оператор. Например, в умножении 5 \* 2 есть два операнда: левый операнд равен 5, а правый операнд равен 2. Иногда их называют «аргументами» вместо «операндов».

**Унарным** называется оператор, который применяется к одному операнду. Например, оператор унарный минус "-" меняет знак числа на противоположный.

**Бинарным** называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме.

Формально, в случае с минусом мы говорим о двух разных операторах, использующих один символ: оператор отрицания (унарный оператор, который обращает знак) и оператор вычитания (бинарный оператор, который вычитает одно число из другого).

Поддерживаются следующие математические операторы:

* Сложение +,
  + если применить к строкам, то он их объединяет в одну
  + если хотя бы один операнд является строкой, то второй будет также преобразован в строку
* Вычитание -,
* Умножение \*,
* Деление /,
* Взятие остатка от деления %,
* Возведение в степень \*\*.

Только бинарный плюс + преобразует операнды в строки. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа.

alert( '6' / '2' ); // 3, оба операнда приводятся к числам

**Приоритет операторов**

Если в выражении есть несколько операторов – порядок их выполнения определяется приоритетом.

Скобки важнее, чем приоритет, скобками меняют порядок выполнения.

Каждый оператор имеет соответствующий номер приоритета. Выполнится раньше тот оператор, у которого число больше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Приоритет унарных операторов выше, чем соответствующих бинарных.

**Присваивание = возвращает значение**

= оператор присваивания. Вызов x = value записывает value в x и возвращает его.

Благодаря этому присваивание можно использовать как часть более сложного выражения:

let a = 1;

let b = 2;

let c = 3 - (a = b + 1);

alert( a ); // 3

alert( c ); // 0

(a = b + 1) будет значение, которое присваивается переменной a (то есть 3). Потом оно используется для дальнейших вычислений.

нужно понимать, как это работает, потому что иногда это можно увидеть в JavaScript-библиотеках.

Писать в таком стиле не рекомендуется. Такой код будет менее понятен и читабелен.

**Присваивание по цепочке**

let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4

Такое присваивание работает справа налево. Сначала вычисляется самое правое выражение 2 + 2, и затем результат присваивается переменным слева: c, b и a. В конце у всех переменных будет одно значение.

Чтобы код читался легче, лучше разделять подобные конструкции на несколько строчек:

c = 2 + 2;

b = c;

a = c;

**Сокращённая арифметика с присваиванием**

Когда нужно применить оператор к переменной и сохранить результат в ней же.

let n = 2;

n += 5; // теперь n = 7 (работает как n = n + 5)

Подобные краткие формы записи существуют для **всех** арифметических и побитовых операторов.

Вызов с присваиванием имеет такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

let n = 2;

n \*= 3 + 5;

alert( n ); // 16 (сначала выполнится правая часть, выражение идентично n \*= 8)

**Инкремент/декремент**

Специальные операторы для увеличения или уменьшения на единицу.

**Инкремент** ++ увеличивает переменную на 1.

**Декремент** -- уменьшает переменную на 1.

let counter = 2;

counter--; // работает как counter = counter - 1

alert( counter ); // 1

Инкремент/декремент можно применить только к переменной. Попытка использовать его на значении, типа 5++, приведёт к ошибке.

* Когда оператор идёт после переменной — это «постфиксная форма»: counter++.
* «Префиксная форма» — это когда оператор идёт перед переменной: ++counter.

Обе эти инструкции делают одно и то же: увеличивают или уменьшают переменную на 1.

Разница между этими операторами в том, что префиксная форма возвращает новое значение, в то время как постфиксная форма возвращает старое (до увеличения/уменьшения числа).

Пример

префиксная форма ++counter увеличивает counter и возвращает новое значение 2. Так что alert покажет 2.

let counter = 1;

let a = ++counter; // (\*)

alert(a); // 2

 постфиксная форма counter++ также увеличивает counter, но возвращает старое значение (которое было до увеличения). Так что alert покажет 1.

let counter = 1;

let a = counter++; // (\*) меняем ++counter на counter++

alert(a); // 1

Подведём итоги:

* Если результат оператора не используется, а нужно только увеличить/уменьшить переменную, тогда без разницы, какую форму использовать:

let counter = 0;

counter++;

++counter;

alert( counter ); // 2, обе строки сделали одно и то же

* Если нужно тут же использовать результат, то нужна префиксная форма:

let counter = 0;

alert( ++counter ); // 1

Если нужно увеличить и при этом получить значение переменной до увеличения – нужна постфиксная форма:

let counter = 0;

console.log(counter++) // 0

console.log(counter) //1

**Инкремент/декремент можно использовать в любых выражениях**

Операторы ++/-- могут также использоваться внутри выражений. Их приоритет выше, чем у большинства других арифметических операций.

Например:

let counter = 1;

alert( 2 \* ++counter ); // 4

Сравните с:

let counter = 1;

alert( 2 \* counter++ ); // 2, потому что counter++ возвращает "старое" значение

Хотя технически здесь всё в порядке, такая запись обычно делает код менее читабельным. Одна строка выполняет множество действий – нехорошо.

При беглом чтении кода можно с лёгкостью пропустить такой counter++, и будет неочевидно, что переменная увеличивается.

Лучше использовать стиль «одна строка – одно действие»:

let counter = 1;

alert( 2 \* counter );

counter++;

**[Побитовые операторы](https://learn.javascript.ru/operators" \l "pobitovye-operatory)**

Побитовые операторы работают с 32-разрядными целыми числами (при необходимости приводят к ним), на уровне их внутреннего двоичного представления.

Они используются редко, когда возникает необходимость оперировать с числами на очень низком (побитовом) уровне.

## [Оператор «запятая»](https://learn.javascript.ru/operators" \l "operator-zapyataya)

Вычисляет несколько выражений, разделяя их запятой ,. Каждое выражение выполняется, но возвращается результат только последнего. **Запятая имеет очень низкий приоритет, ниже чем у присваивания.**

a = 1 + 2, 3 + 4;

alert(a); // 3

a = (1 + 2, 3 + 4);

alert(a); // 7

При use strict выполняется только в скобках, без скобок вылетает ошибка.

Зачем нам оператор, который отбрасывает всё, кроме последнего выражения?

Иногда его используют в составе более сложных конструкций, чтобы сделать несколько действий в одной строке.

Например:

// три операции в одной строке

for (a = 1, b = 3, c = a \* b; a < 10; a++) {

...

}

Такие трюки используются во многих JavaScript-фреймворках. Вот почему мы упоминаем их. Но обычно они не улучшают читабельность кода, поэтому стоит хорошо подумать, прежде чем их использовать.