**Операторы**

### Термины: «унарный», «бинарный», «операнд»

*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:

let x = 1;

x = -x;

alert( x ); // -1, применили унарный минус

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

let x = 1, y = 3;

alert( y - x ); // 2, бинарный минус

### Сложение строк, бинарный +

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну:

let s = "моя" + "строка";

alert(s); // моястрока

Обратите внимание, если хотя бы один операнд является строкой, то второй будет также преобразован к строке.

Например:

alert( '1' + 2 ); // "12"

alert( 2 + '1' ); // "21"

Причём не важно, справа или слева находится операнд-строка. Правило простое: если хотя бы один из операндов является строкой, то второй будет также преобразован к строке.

Тем не менее, помните, что операции выполняются слева направо. Если перед строкой идут два числа, то числа будут сложены перед преобразованием в строку:

alert(2 + 2 + '1' ); // будет "41", а не "221"

Сложение и преобразование строк – это особенность бинарного плюса +. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа. Например, вычитание и деление:

alert( 2 - '1' ); // 1

alert( '6' / '2' ); // 3

### Преобразование к числу, унарный плюс +

Плюс + существует в двух формах: бинарной, которую мы использовали выше, и унарной.

Унарный, то есть применённый к одному значению, плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число.

alert( +true ); // 1

alert( +"" ); // 0

На самом деле это то же самое, что и Number(...), только короче.

Необходимость преобразовывать строки в числа возникает очень часто. Например, обычно значения полей HTML-формы – это строки. А что, если их нужно, к примеру, сложить?

Бинарный плюс сложит их как строки:

let apples = "2";

let oranges = "3";

alert( apples + oranges ); // "23", так как бинарный плюс складывает строки

Поэтому используем унарный плюс, чтобы преобразовать к числу:

let apples = "2";

let oranges = "3";

// оба операнда предварительно преобразованы в числа

alert( +apples + +oranges ); // 5

// более длинный вариант

// alert( Number(apples) + Number(oranges) ); // 5

### Приоритет операторов

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется *приоритетом*, или, другими словами, существует определённый порядок выполнения операторов.

Из школы мы знаем, что умножение в выражении 1 + 2 \* 2 выполнится раньше сложения. Это как раз и есть «приоритет». Говорят, что умножение имеет более высокий приоритет, чем сложение.

Скобки важнее, чем приоритет, так что если мы не удовлетворены порядком по умолчанию, мы можем использовать их, чтобы изменить приоритет. Например, написать (1 + 2) \* 2.

В JavaScript много операторов. Каждый оператор имеет соответствующий номер приоритета. Тот, у кого это число больше – выполнится раньше. Если приоритет одинаковый, то порядок выполнения – слева направо.

Отрывок из [таблицы приоритетов](https://developer.mozilla.org/ru/JavaScript/Reference/operators/operator_precedence) (нет необходимости всё запоминать, обратите внимание, что у унарных операторов приоритет выше, чем у соответствующих бинарных):

|  |  |  |
| --- | --- | --- |
| **Приоритет** | **Название** | **Обозначение** |
| … | … | … |
| 16 | унарный плюс | + |
| 16 | унарный минус | - |
| 14 | умножение | \* |
| 14 | деление | / |
| 13 | сложение | + |
| 13 | вычитание | - |
| … | … | … |
| 3 | присваивание | = |
| … | … | … |

Так как «унарный плюс» имеет приоритет 16, который выше, чем 13 у «сложения» (бинарный плюс), то в выражении "+apples + +oranges" сначала выполнятся унарные плюсы, а затем сложение.

### Присваивание

У присваивания один из самых низких приоритетов: 3. Именно поэтому, когда переменной что-либо присваивают, например, x = 2 \* 2 + 1, то сначала выполнится арифметика, а уже затем произойдёт присваивание =.

let x = 2 \* 2 + 1;

alert( x ); // 5

Возможно 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 оно присваивается переменным слева: c, b и a. В конце у всех переменных будет одно значение. Оператор "=" возвращает значение

Все операторы возвращают значение. Для некоторых это очевидно, например сложение + или умножение \*. Но и оператор присваивания не является исключением.

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

В примере выше результатом (a = b + 1) будет значение, которое присваивается в a (то есть 3). Потом оно используется для дальнейших вычислений.

Забавное применение присваивания, не так ли? Нам нужно понимать, как это работает, потому что иногда это можно увидеть в JavaScript-библиотеках, но писать самим в таком стиле не рекомендуется. Такие трюки не сделают ваш код более понятным или читабельным.

### Остаток от деления %

Оператор взятия остатка %, несмотря на обозначение, никакого отношения к процентам не имеет.

Его результат a % b – это остаток от деления a на b.

Например:

alert( 5 % 2 ); // 1, остаток от деления 5 на 2

### Возведение в степень \*\*

Оператор возведения в степень \*\* недавно добавили в язык.

Для натурального числа b результат a \*\* b равен a, умноженному на само себя b раз.

Например:

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

alert( 2 \*\* 3 ); // 8 (2 \* 2 \* 2)

Оператор работает и для нецелых чисел.

Например:

alert( 4 \*\* (1/2) ); // 2 (степень 1/2 эквивалентна взятию квадратного корня)

alert( 8 \*\* (1/3) ); // 2 (степень 1/3 эквивалентна взятию кубического корня)

### Инкремент/декремент

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу.

Для этого существуют даже специальные операторы:

* **Инкремент** ++ увеличивает на 1:
* let counter = 2;
* counter++; // работает как counter = counter + 1, просто запись короче

alert( counter ); // 3

**Декремент** -- уменьшает на 1:

let counter = 2;

counter--; // работает как counter = counter - 1, просто запись короче

alert( counter ); // 1

Важно:

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

Операторы ++ и -- могут быть расположены не только после, но и до переменной.

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

Обе эти формы записи делают одно и то же: увеличивают counter на 1.

Есть ли разница между ними? Да, но увидеть её мы сможем, только если будем использовать значение, которое возвращают ++/--.

Префиксная форма возвращает новое значение, в то время как постфиксная форма возвращает старое (до увеличения/уменьшения числа).

Чтобы увидеть разницу, вот небольшой пример:

let counter = 1;

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

alert(a); // 2

В строке (\*) *префиксная* форма увеличения counter, она возвращает новое значение 2. Так что alert покажет 2.

Теперь посмотрим на постфиксную форму:

let counter = 1;

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

alert(a); // 1

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

Подведём итоги:

* Если результат оператора не используется, а нужно только увеличить/уменьшить переменную – без разницы, какую форму использовать:
* let counter = 0;
* counter++;
* ++counter;

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

Если хочется 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* – постфиксная форма:

let counter = 0;

alert( counter++ ); // 0

Инкремент/декремент можно использовать в любых выражениях

Операторы ++/-- могут также использоваться внутри выражений. Их приоритет выше, чем у 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 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++;

### Сокращённая арифметика с присваиванием

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

Например:

let n = 2;

n = n + 5;

n = n \* 2;

Эту запись можно укоротить при помощи совмещённых операторов += и \*=:

let n = 2;

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

n \*= 2; // теперь n=14 (работает как n = n \* 2)

alert( n ); // 14

Подобные краткие формы записи существуют для всех арифметических и побитовых операторов: /=, -= и так далее.

Вызов с присваиванием имеет в точности 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, превратив выражение в n \*= 8)

### Оператор запятая

Оператор «запятая», редко используется и является одним из самых необычных. Иногда он используется для написания более короткого кода, поэтому нам нужно знать его, чтобы понимать, что при этом происходит.

Оператор запятая предоставляет нам возможность вычислять несколько выражений, разделяя их запятой ,. Каждое выражение выполняется, но возвращается результат только последнего.

Например:

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

alert( a ); // 7 (результат 3 + 4)

Первое выражение 1 + 2 выполняется, а результат отбрасывается. Затем идёт 3 + 4, выражение выполняется и возвращается результат.

Запятая имеет очень низкий приоритет

Оператор , имеет очень низкий приоритет, ниже =, поэтому скобки важны в приведённом выше примере.

Без них в a = 1 + 2, 3 + 4 сначала выполнится +, суммируя числа в a = 3, 7, затем оператор присваивания = присвоит a = 3, а то, что идёт дальше, будет игнорировано. Всё так же, как в (a = 1 + 2), 3 + 4.

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.