14 января 2023 г.

Базовые операторы, математика

Многие операторы знакомы нам ещё со школы: сложение `+`, умножение `*`, вычитание `-` и так далее.

В этой главе мы начнём с простых операторов, а потом сконцентрируемся на специфических для JavaScript аспектах, которые не проходят в школьном курсе арифметики.

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

Прежде, чем мы двинемся дальше, давайте разберёмся с терминологией.

- *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 вычитает значения
```

Формально, в последних примерах 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).

Математика

Поддерживаются следующие математические операторы:

- Сложение `+`,
- Вычитание `-`,
- Умножение `*`,
- Деление `/`,
- Взятие остатка от деления `%`,
- Возведение в степень `**`.

Первые четыре оператора очевидны, а про `%` и `**` стоит сказать несколько слов.

Взятие остатка %

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

Результат `a % b` – это остаток от целочисленного деления `a` на `b`.

Например:

```
1  alert( 5 % 2 ); // 1, остаток от деления 5 на 2
2  alert( 8 % 3 ); // 2, остаток от деления 8 на 3
3  alert( 8 % 4 ); // 0, остаток от деления 8 на 4
```

Возведение в степень **

Оператор возведения в степень `a ** b` возводит `a` в степень `b`.

В школьной математике мы записываем это как a^b.

Например:

```
1  alert( 2 ** 2 ); // 2² = 4
2  alert( 2 ** 3 ); // 2³ = 8
3  alert( 2 ** 4 ); // 2⁴ = 16
```

Математически, оператор работает и для нецелых чисел. Например, квадратный корень является возведением в степень `1/2`:

```
1  alert( 4 ** (1/2) ); // 2 (степень 1/2 эквивалентна взятию квадратного корня)
2  alert( 8 ** (1/3) ); // 2 (степень 1/3 эквивалентна взятию кубического корня)
```

Сложение строк при помощи бинарного +

Давайте рассмотрим специальные возможности операторов JavaScript, которые выходят за рамки школьной арифметики.

Обычно при помощи плюса `'+'` складывают числа.

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

```
1  let s = "моя" + "строка";
2  alert(s); // моястрока
```

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

Например:

```
alert( '1' + 2 ); // "12"
alert( 2 + '1' ); // "21"
```

Как видите, не важно, первый или второй операнд является строкой.

Вот пример посложнее:

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

Здесь операторы работают один за другим. Первый `+` складывает два числа и возвращает `4`, затем следующий `+` объединяет результат со строкой, производя действие `4 + '1' = '41'`.

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( 6 - '2' ); // 4, '2' приводится к числу
alert( '6' / '2' ); // 3, оба операнда приводятся к числам
```

Приведение к числу, унарный +

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.

Например:

```
// Не влияет на числа
let x = 1;
alert( +x ); // 1

let y = -2;
alert( +y ); // -2

// Преобразует не числа в числа
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 объединяет 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
```

С точки зрения математика, такое изобилие плюсов выглядит странным. Но с точки зрения программиста тут нет ничего особенного: сначала выполнятся унарные плюсы, которые приведут строки к числам, а затем бинарный `'+'` их сложит.

Почему унарные плюсы выполнились до бинарного сложения? Как мы сейчас увидим, дело в их приоритете.

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 (нет необходимости всё запоминать, обратите внимание, что приоритет унарных операторов выше, чем соответствующих бинарных):

Приоритет	Название	Обозначение
...	...	...
15	унарный плюс	+
15	унарный минус	-
14	возведение в степень	**
13	умножение	*
13	деление	/
12	сложение	+
12	вычитание	-
...	...	...
2	присваивание	=
...	...	...

Так как «унарный плюс» имеет приоритет `15`, который выше, чем `12` у «сложения» (бинарный плюс), то в выражении `"+apples + +oranges"` сначала выполнятся унарные плюсы, а затем сложение.

Присваивание

Давайте отметим, что в таблице приоритетов также есть оператор присваивания `=`. У него один из самых низких приоритетов: `2`.

Именно поэтому, когда переменной что-либо присваивают, например, `x = 2 * 2 + 1`, то сначала выполнится арифметика, а уже затем произойдёт присваивание `=` с сохранением результата в `x`.

```
let x = 2 * 2 + 1;

alert( x ); // 5
```

Присваивание = возвращает значение

Тот факт, что `=` является оператором, а не «магической» конструкцией языка, имеет интересные последствия.

Большинство операторов в JavaScript 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 переменной `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.

Однако писать самим в таком стиле не рекомендуется. Такие трюки не сделают ваш код более понятным или читабельным.

Присваивание по цепочке

Рассмотрим ещё одну интересную возможность: цепочку присваиваний.

```
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.

Опять-таки, чтобы код читался легче, лучше разделять подобные конструкции на несколько строчек:

```
c = 2 + 2;
b = c;
a = c;
```

Польза от такого стиля особенно ощущается при быстром просмотре кода.

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, выражение идентично n *=
```

Инкремент/декремент

Одной из наиболее частых числовых операций является увеличение или уменьшение на единицу.

Для этого существуют даже специальные операторы:

- **Инкремент** `++` увеличивает переменную на 1:

```
let counter = 2;
counter++;        // работает как counter = counter + 1, просто запись коро
alert( counter ); // 3
```

- **Декремент** `--` уменьшает переменную на 1:

```
let counter = 2;
counter--;        // работает как counter = counter - 1, просто запись коро
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 инструкции 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 `++/--` .

Давайте проясним этот момент. Как мы знаем, все операторы возвращают значение. Операторы инкремента/декремента не исключение. 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 `++counter` увеличивает `counter` и 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, тогда 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* – нужна 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 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" зна
```

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

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

Эти операторы не являются чем-то специфичным для JavaScript, они поддерживаются в большинстве языков программирования.

Поддерживаются следующие побитовые операторы:

- AND(и) (`&`)
- OR(или) (`|`)
- XOR(побитовое исключающее или) (`^`)
- NOT(не) (`~`)
- LEFT SHIFT(левый сдвиг) (`<<`)
- RIGHT SHIFT(правый сдвиг) (`>>`)
- ZERO-FILL RIGHT SHIFT(правый сдвиг с заполнением нулями) (`>>>`)

Они используются редко, когда возникает необходимость оперировать с числами на очень низком (побитовом) уровне. В ближайшем времени они нам не понадобятся, так как веб-разработчики редко к ним прибегают, хотя в некоторых сферах (например, в криптографии) они полезны. Можете прочитать раздел о них на MDN, когда возникнет реальная необходимость.

Оператор «запятая»

Оператор «запятая» (`,`) редко применяется 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:

```
1  let a = (1 + 2, 3 + 4);
2
3  alert( a ); // 7 (результат вычисления 3 + 4)
```

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

> **Запятая имеет очень низкий приоритет**
>
> Пожалуйста, обратите внимание, что 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.

Задачи

Постфиксная и префиксная формы

важность: 5

Чему будут равны переменные `a`, `b`, `c` и `d` в примере ниже?

```
let a = 1, b = 1;

let c = ++a; // ?
let d = b++; // ?
```

решение

Результат присваивания

важность: 3

Чему будут равны переменные `a` и `x` после исполнения кода в примере ниже?

```
let a = 2;

let x = 1 + (a *= 2);
```

решение

Преобразование типов

важность: 5

Какой результат будет у выражений ниже?

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
```

```
"4" - 2
"4px" - 2
"  -9  " + 5
"  -9  " - 5
null + 1
undefined + 1
" \t \n" - 2
```

Подумайте как следует, запишите ответы и сверьтесь с решением.

решение

Исправьте сложение

важность: 5

Ниже приведён код, который запрашивает у пользователя два числа и показывает их сумму.

Он работает неправильно. Код в примере выводит `12` (для значения полей по умолчанию).

В чём ошибка? Исправьте её. Результат должен быть `3`.

```
let a = prompt("Первое число?", 1);
let b = prompt("Второе число?", 2);

alert(a + b); // 12
```

решение

Предыдущий урок | Следующий урок

Поделиться

Карта учебника

Комментарии

- Если вам кажется, что в статье что-то не так - вместо комментария напишите на GitHub.
- Для одной строки кода используйте тег `<code>`, для нескольких строк кода — тег `<pre>`, если больше 10 строк — ссылку на песочницу (plnkr, JSBin, codepen...)
- Если что-то непонятно в статье — пишите, что именно и с какого места.