JS - операторы

Базовые операторы и математика, операторы сравнения, логические операторы и операторы нулевого слияния

Работа с переменными (запись переменных)

в JavaScript вы можете записывать переменные через запятую при объявлении переменных с помощью ключевого слова var, let или const

```
var a = 1, b = 2, c = 3;
let x = 10, y = 20, z = 30;
const PI = 3.14, GRAVITY = 9.8;
```

Деструктуризация массивов

```
let [a, b, c] = [1, 2, 3];
```

Деструктуризация объектов

```
let {x, y, z} = {x: 10, y: 20, z: 30};
```

Эти способы удобны, когда вам нужно извлечь значения из массива или объекта и присвоить их переменным в одной строке.

Базовые операторы и математика

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 вычитает значения
```

+ в JS со строками

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

Сложение строк при помощи бинарного + (concat - конкатенация)
Обычно при помощи плюса '+' складывают числа.
Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну. Обратите внимание, если хотя бы один операнд является строкой, то второй будет также преобразован в строку. Сложение и преобразование строк — это особенность бинарного плюса +. Другие арифметические операторы работают только с числами и всегда преобразуют операнды в числа.

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

Приведение к числу, унарный +

Унарный, то есть примененный к одному значению, плюс + ничего не делает с числами. Но если операнд не число, унарный плюс преобразует его в число. На самом деле это то же самое, что и Number(...), только короче.

```
// 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
```

```
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
```

Математика

Поддерживаются следующие математические операторы:

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

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.

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

Приоритет операторов

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

Скобки важнее, чем приоритет, так что, если мы не удовлетворены порядком по умолчанию, мы можем использовать их, чтобы изменить приоритет.

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.

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

Присваивание

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 происходит присваивание = с сохранением результата в x.

Присваивание = возвращает значение
Вызов *x = value* записывает value в x и возвращает его.

Присваивание по цепочке
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.

```
let a = 1;
let b = 2;

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

alert( a ); // 3
alert( c ); // 0

let a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4
alert( b ); // 4
alert( c ); // 4
```

Сокращенная арифметика с присваиванием

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

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 = n + 5;
n = n * 2;
```

```
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
```

Инкремент/декремент

Одной из наиболее частых числовых операций является увеличение или уменьшение на единицу.
Для этого существуют даже специальные операторы: “++” и “--”

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.

- префиксная форма ++counter увеличивает counter и возвращает новое значение 2
- постфиксная форма counter++ также увеличивает counter, но возвращает старое значение

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

```
let counter = 2;
counter--;        // работает как counter = counter - 1
alert( counter ); // 1
```

```
let counter = 1;
let a = ++counter; // (*)
```

```
let counter = 1;
let a = counter++; // (*) меняем ++counter на counter++

alert(a); // 1
```

Операторы сравнения

```
alert( 2 > 1 );  // true (верно)
alert( 2 == 1 ); // false (неверно)
alert( 2 != 1 ); // true (верно)
```

- Больше/меньше: a > b, a < b.
- Больше/меньше или равно: a >= b, a <= b.
- Равно: a == b. Обратите внимание, для сравнения используется двойной знак равенства ==. Один знак равенства a = b означал бы присваивание.
- Не равно. В математике обозначается символом ≠, но в JavaScript записывается как a != b.

Все операторы сравнения возвращают значение логического типа:

- true – означает «да», «верно», «истина».
- false – означает «нет», «неверно», «ложь».

Сравнение строк

Чтобы определить, что одна строка больше другой, JavaScript использует «алфавитный» или «лексикографический» порядок.
Другими словами, строки сравниваются посимвольно.

Алгоритм сравнения двух строк довольно прост:

- Сначала сравниваются первые символы строк.
- Если первый символ первой строки больше (меньше), чем первый символ второй, то первая строка больше (меньше) второй. Сравнение завершено.
- Если первые символы равны, то таким же образом сравниваются уже вторые символы строк.
- Сравнение продолжается, пока не закончится одна из строк.
- Если обе строки заканчиваются одновременно, то они равны. Иначе, большей считается более длинная строка.

```
alert( 'Я' > 'А' ); // true
alert( 'Коты' > 'Кода' ); // true
alert( 'Сонный' > 'Сон' ); // true
```

В примерах выше сравнение 'Я' > 'А' завершится на первом шаге, тогда как строки 'Коты' и 'Кода' будут сравниваться посимвольно:

1. К равна К.
2. о равна о.
3. т больше, чем д. На этом сравнение заканчивается. Первая строка больше.

Сравнение разных типов

При сравнении значений разных типов JavaScript приводит каждое из них к числу.

Логическое значение true становится 1, а false – 0.

Забавное следствие

Возможна следующая ситуация:

- Два значения равны.
- Одно из них `true` как логическое значение, другое – `false`.

Например:

```
1  let a = 0;
2  alert( Boolean(a) ); // false
3
4  let b = "0";
5  alert( Boolean(b) ); // true
6
7  alert(a == b); // true!
```

С точки зрения JavaScript, результат ожидаем. Равенство преобразует значения, используя числовое преобразование, поэтому `"0"` становится `0`. В то время как явное преобразование с помощью `Boolean` использует другой набор правил.

Строгое сравнение

Использование обычного сравнения == может вызывать проблемы. Например, оно не отличает 0 от false.

Оператор строгого равенства === проверяет равенство без приведения типов.

Другими словами, если a и b имеют разные типы, то проверка a === b немедленно возвращает false без попытки их преобразования.

Ещё есть оператор строгого неравенства !==, аналогичный !=.

Оператор строгого равенства дольше писать, но он делает код более очевидным и оставляет меньше места для ошибок.

```
alert( 0 == false ); // true
```

```
alert( '' == false ); // true
```

```
alert( 0 === false ); // false, так как сравниваются разные типы
```

Сравнение с null и undefined

Поведение null и undefined при сравнении с другими значениями — особое:

- При строгом равенстве ===
  Эти значения различны, так как различны их типы.
- При нестрогом равенстве ==
  Эти значения равны друг другу и не равны никаким другим значениям. Это специальное правило языка.

При использовании математических операторов и других операторов сравнения < > <= >=

Значения null/undefined преобразуются к числам: null становится 0, а undefined – NaN.

Посмотрим, какие забавные вещи случаются, когда мы применяем эти правила. И, что более важно, как избежать ошибок при их использовании.

```
alert( null == undefined ); // true

alert( null === undefined ); // false
```

Странный результат сравнения null и 0

Сравним null с нулём:

С точки зрения математики это странно. Результат последнего сравнения говорит о том, что "null больше или равно нулю", тогда результат одного из сравнений выше должен быть true, но они оба ложны.

Причина в том, что нестрогое равенство и сравнения > < >= <= работают по-разному. Сравнения преобразуют null в число, рассматривая его как 0. Поэтому выражение (3) null >= 0 истинно, а null > 0 ложно.

С другой стороны, для нестрогого равенства == значений undefined и null действует особое правило: эти значения ни к чему не приводятся, они равны друг другу и не равны ничему другому. Поэтому (2) null == 0 ложно.

```
alert( null > 0 );  // (1) false
alert( null == 0 ); // (2) false
alert( null >= 0 ); // (3) true
```

Несравненное значение undefined

Значение undefined несравнимо с другими значениями

Почему же сравнение undefined с нулём всегда ложно?

На это есть следующие причины:

- Сравнения (1) и (2) возвращают false, потому что undefined преобразуется в NaN, а NaN – это специальное числовое значение, которое возвращает false при любых сравнениях.
- Нестрогое равенство (3) возвращает false, потому что undefined равно только null, undefined и ничему больше.

```
alert( undefined > 0 ); // false (1)
alert( undefined < 0 ); // false (2)
alert( undefined == 0 ); // false (3)
```

Как избежать проблем

Относитесь очень осторожно к любому сравнению с undefined/null, кроме случаев строгого равенства ===.

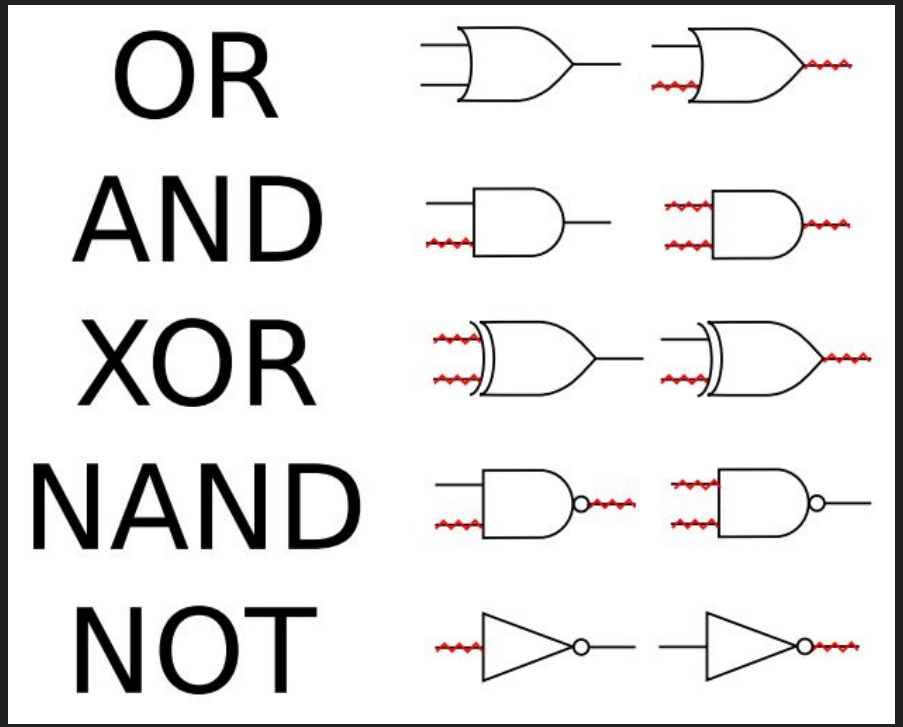
Не используйте сравнения >= > < <= с переменными, которые могут принимать значения null/undefined, разве что вы полностью уверены в том, что делаете. Если переменная может принимать эти значения, то добавьте для них отдельные проверки.

Логические операторы

В JavaScript есть семь логических операторов:

- || (ИЛИ)
- ||= (Оператор логического присваивания ИЛИ)
- && (И)
- &&= (Оператор логического присваивания И)
- ! (НЕ)
- ?? (Оператор нулевого слияния)
- ??= (Оператор нулевого присваивания)

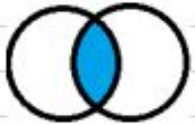
В жизни

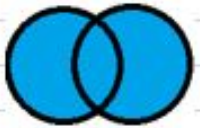


A	B	A and B	A or B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

коньюнкция
и
&
лог. умножение

дизьюнкция
или
|
лог. сложение





|| (ИЛИ)

Традиционно в программировании ИЛИ предназначено только для манипулирования булевыми значениями: в случае, если какой-либо из аргументов true, он вернёт true, в противоположной ситуации возвращается false.

ИЛИ "||" находит первое истинное значение

Оператор || выполняет следующие действия:

- Вычисляет операнды слева направо.
- Каждый операнд конвертирует в логическое значение. Если результат true, останавливается и возвращает исходное значение этого операнда.
- Если все операнды являются ложными (false), возвращает последний из них.
- Значение возвращается в исходном виде, без преобразования.

Другими словами, цепочка ИЛИ || возвращает первое истинное значение или последнее, если такое значение не найдено.

```
alert( true || true );   // true
alert( false || true );  // true
alert( true || false );  // true
alert( false || false ); // false
```

```
let hour = 12;
let isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend) {
  alert( 'Офис закрыт.' ); // это выходной
}
```

||= (Оператор логического присваивания ИЛИ)

Оператор логического присваивания ИЛИ ||= записывается как обычный ИЛИ || с добавлением символа присваивания =. Такая запись не случайна, так как результат выполнения данного оператора напрямую зависит от действий уже известного нам ||.

Оператор ||= принимает два операнда и выполняет следующие действия:

- Вычисляет операнды слева направо.
- Конвертирует a в логическое значение.
- Если a ложно, присваивает a значение b.

На практике почти не используется так как усложняет чтение кода

&& (И)

В традиционном программировании И возвращает true, если оба аргумента истинны, а иначе – false.

И «&&» находит первое ложное значение

Оператор && выполняет следующие действия:

- Вычисляет операнды слева направо.
- Каждый операнд преобразует в логическое значение. Если результат false, останавливается и возвращает исходное значение этого операнда.
- Если все операнды были истинными, возвращается последний.
- Другими словами, И возвращает первое ложное значение. Или последнее, если ничего не найдено.

Вышеуказанные правила схожи с поведением ИЛИ. Разница в том, что И возвращает первое ложное значение, а ИЛИ– первое истинное.

Приоритет оператора && больше, чем у ||
Приоритет оператора И && больше, чем ИЛИ ||, так что он выполняется раньше. Таким образом, код a && b || c && d по существу такой же, как если бы выражения && были в круглых скобках: (a && b) || (c && d).

```
alert( true && true );   // true
alert( false && true );  // false
alert( true && false );  // false
alert( false && false ); // false
```

```
let hour = 12;
let minute = 30;

if (hour == 12 && minute == 30) {
  alert( 'Время 12:30' );
}
```

&&= (Оператор логического присваивания И)

Принцип действия &&= практически такой же, как и у оператора логического присваивания ИЛИ ||=. Единственное отличие заключается в том, что &&= присвоит a значение b только в том случае, если a истинно.

На практике, в отличие от ||=, оператор &&= используется достаточно редко – обычно, в комбинации с более сложными языковыми конструкциями, о которых мы будем говорить позже. Подобрать контекст для применения данного оператора – довольно непростая задача.

На практике не используется

! (НЕ)

Оператор принимает один аргумент и выполняет следующие действия:

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

В частности, двойное НЕ !! используют для преобразования значений к логическому типу
То есть первое НЕ преобразует значение в логическое значение и возвращает обратное, а второе НЕ снова инвертирует его. В конце мы имеем простое преобразование значения в логическое.

Приоритет НЕ ! является наивысшим из всех логических операторов, поэтому он всегда выполняется первым, перед && или ||.

```
alert( !true ); // false
alert( !0 ); // true
```

```
alert( !!"непустая строка" ); // true
alert( !!null ); // false
```

```
alert( Boolean("непустая строка") ); // true
alert( Boolean(null) ); // false
```

Оператор нулевого слияния (??)

Оператор нулевого слияния представляет собой два вопросительных знака ??.

Так как он обрабатывает null и undefined одинаковым образом, то для этой статьи мы введем специальный термин. Для краткости будем говорить, что значение «определено», если оно не равняется ни null, ни undefined.

Результат выражения a ?? b будет следующим:

- если a определено, то a,
- если a не определено, то b.

Иначе говоря, оператор ?? возвращает первый аргумент, если он не null/undefined, иначе второй.

Важное различие между ними заключается в том, что:

- || возвращает первое истинное значение.
- ?? возвращает первое определённое значение.

Проще говоря, оператор || не различает false, 0, пустую строку "" и null/undefined. Для него они все одинаковы, т.е. являются ложными значениями. Если первым аргументом для оператора || будет любое из перечисленных значений, то в качестве результата мы получим второй аргумент.

Однако на практике часто требуется использовать значение по умолчанию только тогда, когда переменная является null/undefined. Ведь именно тогда значение действительно неизвестно/не определено.

По соображениям безопасности JavaScript запрещает использование оператора ?? вместе с && и ||, если приоритет явно не указан при помощи круглых скобок.

```
let height = 0;

alert(height || 100); // 100
alert(height ?? 100); // 0
```

```
let x = 1 && 2 ?? 3; // Синтаксическая ошибка
```

```
let x = (1 && 2) ?? 3; // Работает без ошибок

alert(x); // 2
```

Оператор нулевого присваивания (??=)

Оператор ??= присвоит x значение y только в том случае, если x не определено (null/undefined).

Обратите внимание: если бы userAge не был равен null/undefined, то выражение справа от ??= никогда бы не выполнилось:

```
let userAge = null;

userAge ??= 18;

alert(userAge) // 18
```

```
let userAge = 18;

userAge ??= alert("не сработает");
userAge ??= 21;
userAge ??= null;

alert(userAge) // по-прежнему 18
```

Ресурсы

Базовые операторы и математика - ТЫК

Таблица приоритетов операторов - ТЫК

Операторы сравнения - ТЫК

Логические операторы - ТЫК

Операторы нулевого слияния - ТЫК