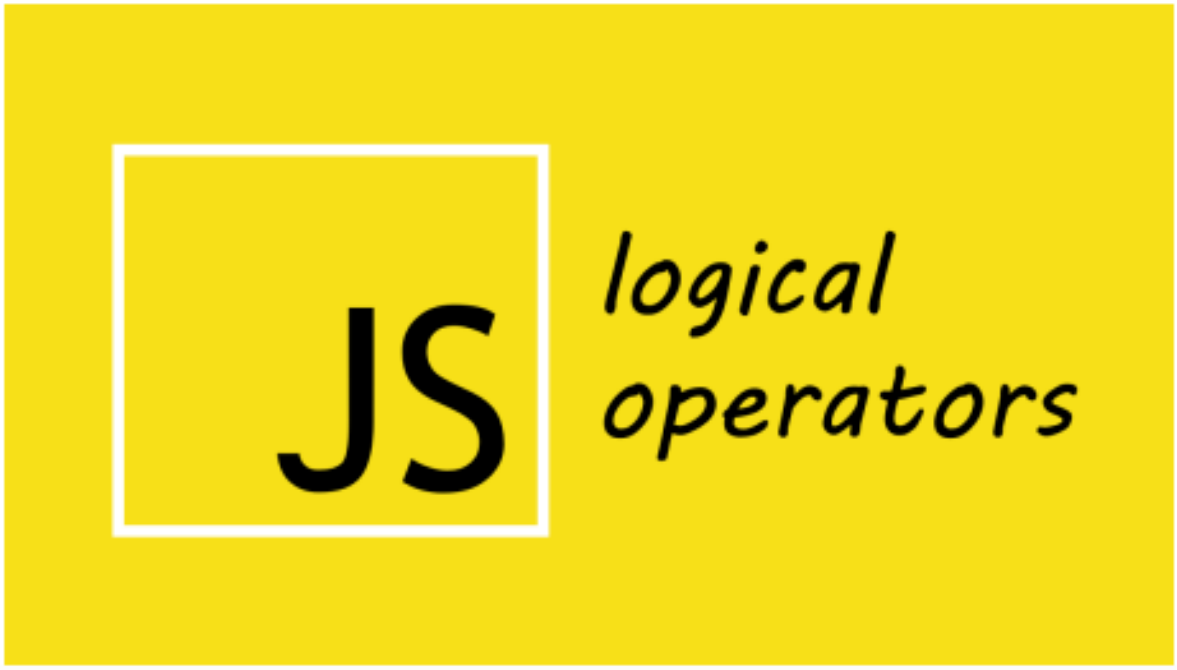
Лабораторная работа №4. Логические и побитовые операторы в JavaScript



Содержание:

На этом занятии мы изучим четыре логических оператора: «НЕ», «ИЛИ», «И» и нулевого слияния. Кроме этого, дополнительно ещё рассмотрим побитовые операторы, которые используются в коде довольно редко, но нужны для реализации криптографических и других алгоритмов, требующих работу с битами.

Логические операторы

В JavaScript четыре логических оператора:

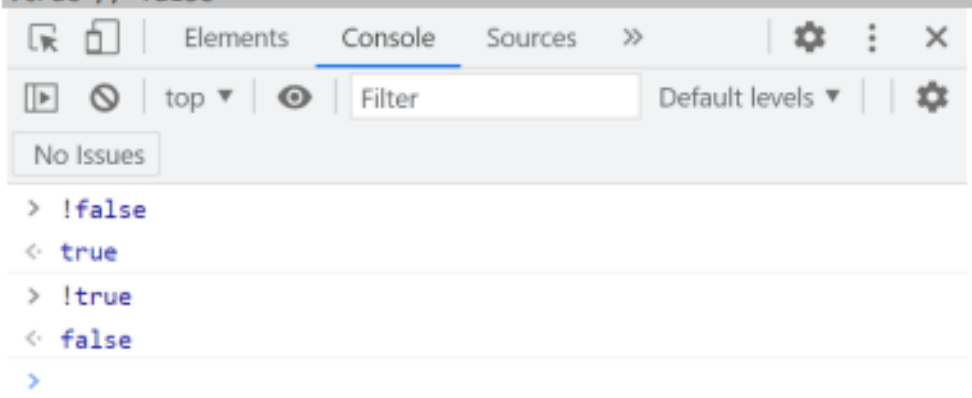
- логическое «НЕ» `!`;
- логическое «И» `&&`;

- логическое «ИЛИ» `||`;
- оператор нулевого слияния `??`.

Оператор «НЕ»

Оператор «НЕ» обозначается в JavaScript с помощью `!`. Он является префиксным унарным оператором, который всегда возвращает значение логического типа, т.е. `true` или `false`.

```
!false // true
!true // false
```



Ложные и истинные значения

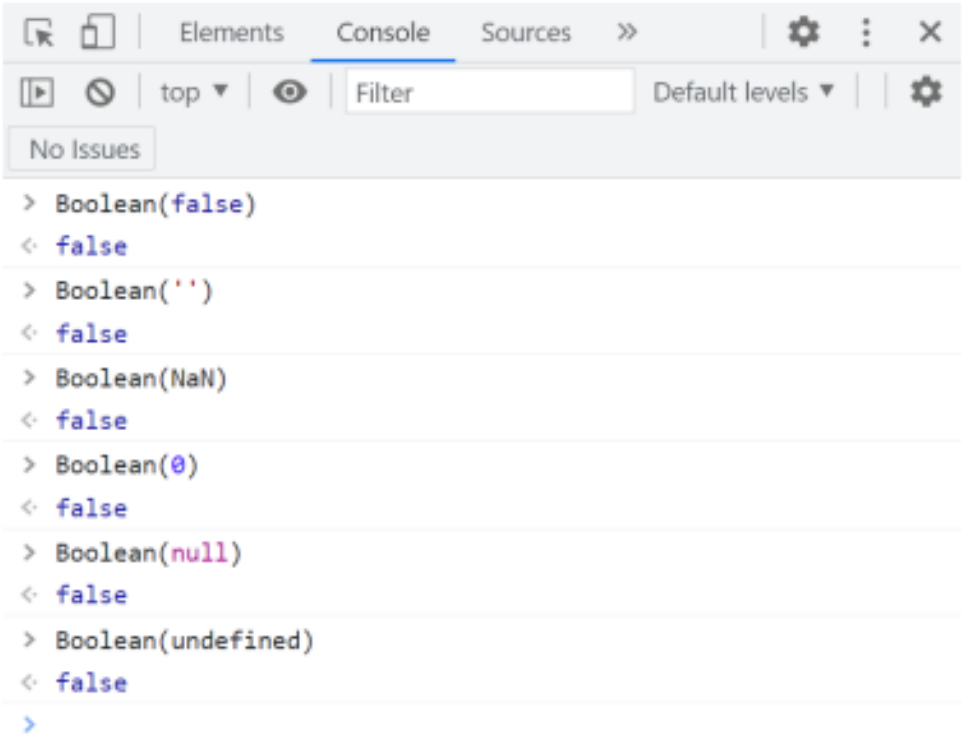
Ложными значениями в JavaScript являются те, которые при приведении к логическому типу дают `false`. Конвертировать любое значение в логическое можно с помощью функции `Boolean()` или двойного оператора «НЕ»:

```
const resultA = Boolean(value);
const resultB = !!value;
```

Для этого `Boolean()` нужно передать в качестве аргумента значение, которое нужно привести к булевому. Если эта функция вернёт `false`, то значение, которое вы ей передели является **ложным**. В противном случае вы на выходе получите `true` и, следовательно, это значение является **истинным**.

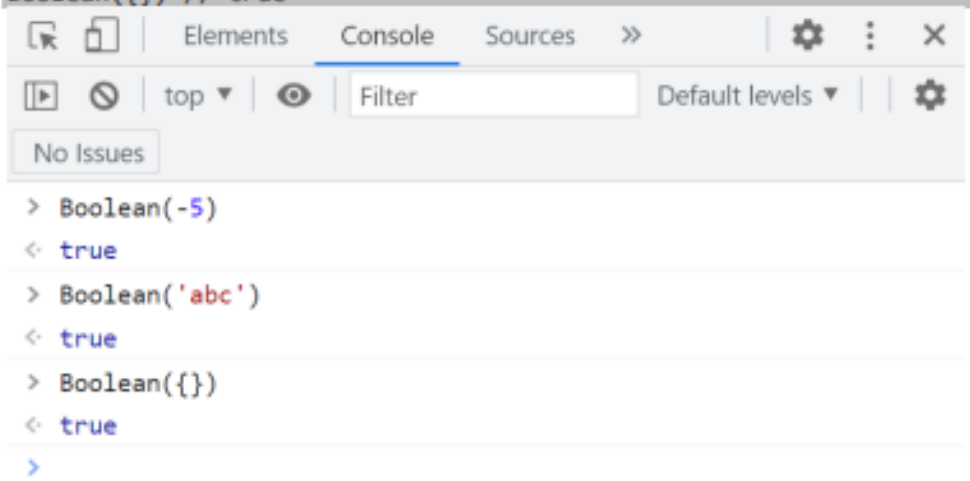
В JavaScript следующие значения являются **ложными**:

- `false` (ложь);
- `""` или `''` (пустая строка);
- `NaN` (специальный числовой тип данных который обозначает «не число»);
- `0` (число ноль);
- `null` («пустое» значение);
- `undefined` («неопределённое» значение).



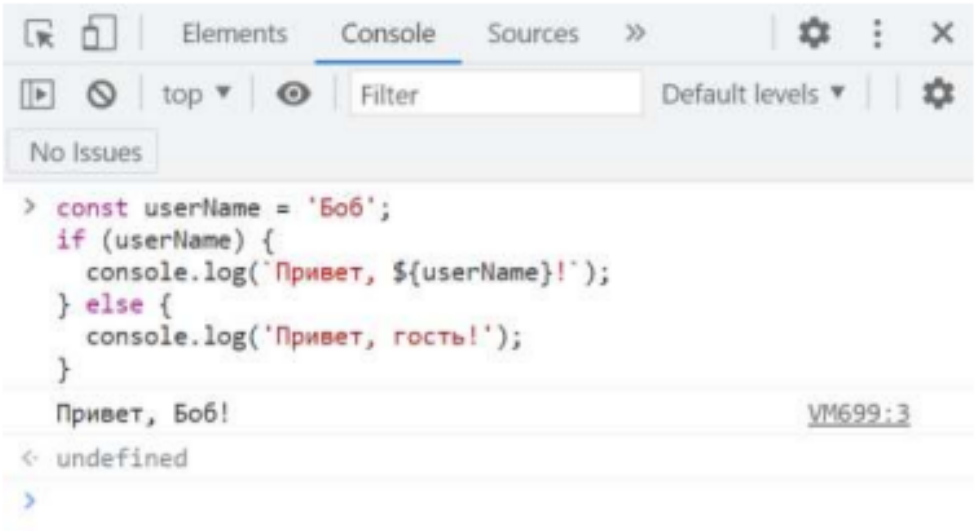
Все остальные кроме этих величин являются **истинными**:

```
// примеры
Boolean(-5) // true
Boolean('abc') // true
Boolean({}) // true
```



Приведение выражения к истинности или лжи применяется, например, в условной инструкции `if`:

```
const userName = 'Боб';
if (userName) {
  console.log(`Привет, ${userName}!`);
} else {
  console.log('Привет, гость!');
}
```



Здесь в качестве условия выступает выражение `userName`. В данном случае оно будет приведено к истине, потому что `Boolean(userName) === true`. В результате в консоли мы уведем сообщение «Привет, Боб!».

Если бы переменная `userName` содержала другое значение, которое приводилась бы к `false`, то в консоли было бы напечатано «Привет, гость!».

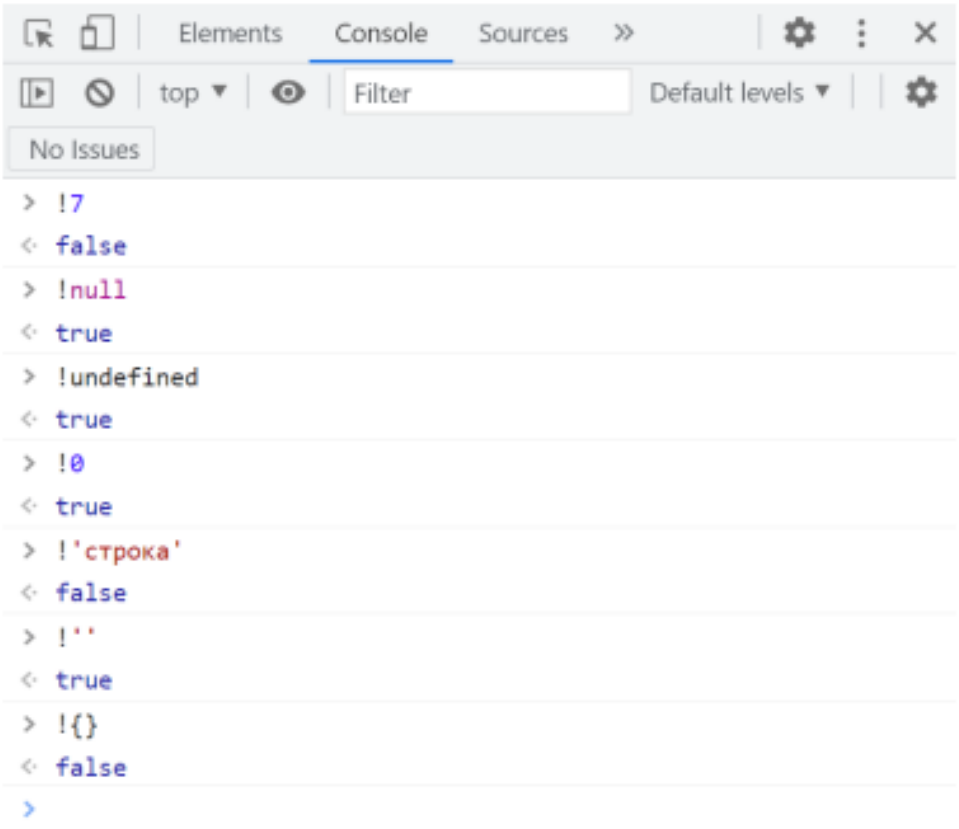
То есть по факту в условии `if` мы делаем следующее:

```
const userName = 'Боб';
if (Boolean(userName)) {
  console.log(`Привет, ${userName}!`);
} else {
  console.log('Привет, гость!');
}
```

Оператор «НЕ» с небулевыми значениями

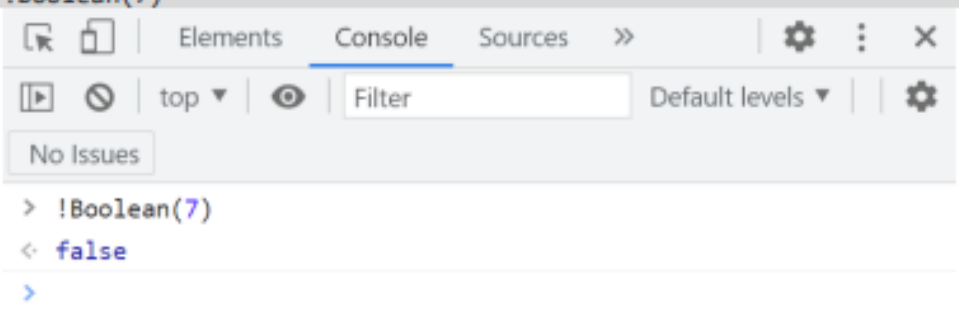
Пример использования `!` с нелогическими величинами:

```
!7 // false
!null // true
!undefined // true
!0 // true
!'строка' // false
!'' // true
!{} // false
```



Понять какой будет результат каждого выражения очень просто, если сначала привести значение, указанное до `!` к логическому значению, а затем выполнить его отрицание. Например, выражение `!7` можно записать так:

```
!Boolean(7)
```



Эти же примеры, но с двойным отрицанием:

```
!!7 // true
!!null // false
!!undefined // false
!!0 // false
!!'строка' // true
!!'' // false
!!{} // true
```

Используя отрицание отрицания, можно легко преобразовать любое значение в `true` или `false`. То есть точно также как с помощью функции `Boolean()`. Это ещё один способ проверить ложность или истинность того или иного значения.

Операторы логического «И» и «ИЛИ»

Операторы `&&` и `||` используются обычно с логическими значениями:

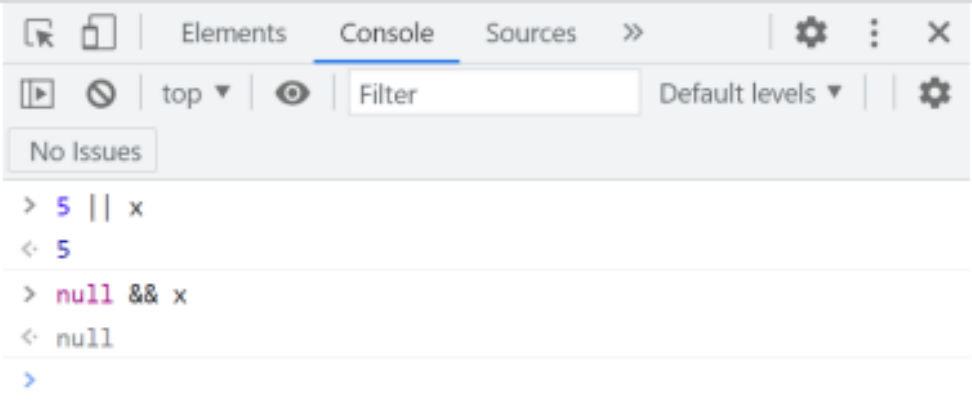
```
false && false // false
true && false // false
false && true // false
true && true // true

false || false // false
true || false // true
false || true // true
true || true // true
```

В этом случае возвращаемое значение таких выражений также будет булевым.

Но на самом деле эти операторы **всегда возвращают значение одного из операндов**. При этом какого именно зависит от их значений. В случае с небулевыми значениями, результат также может быть не булевым.

Очень важный момент заключается в том, что эти операторы имеют **сокращенный способ вычисления (short-circuiting)**. Это означает, что **если результат уже известен, то следующий операнд уже не вычисляется**. Он просто игнорируется.

```
5 || x // сразу 5, x не вычисляется, т.к. результат и так уже понятен
null && x // сразу null, x не вычисляется, т.к. результат и так уже понятен
```



Как вычисляется выражение с «&&»

Пример выражения с оператором «логическое И»:

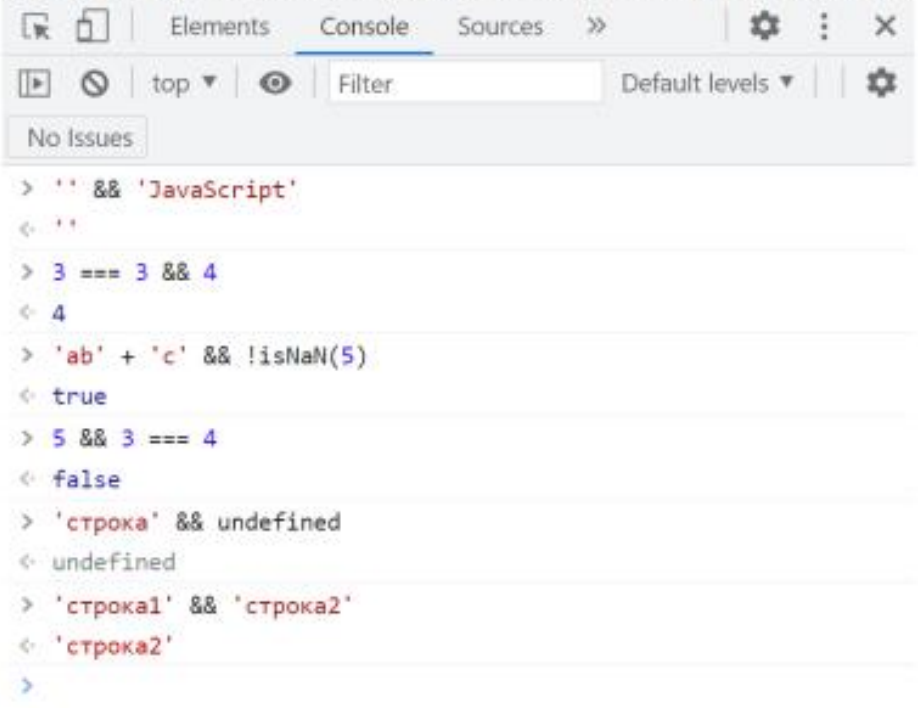
```
exprA && exprB
```

Если выражение `exprA` ложно, то `exprB` игнорируется и возвращается результат `exprA` как результат всего этого выражения. То есть, интерпретатор не

рассматривает дальше это выражение, если exprA ложно, он сразу же возвращает его результат.

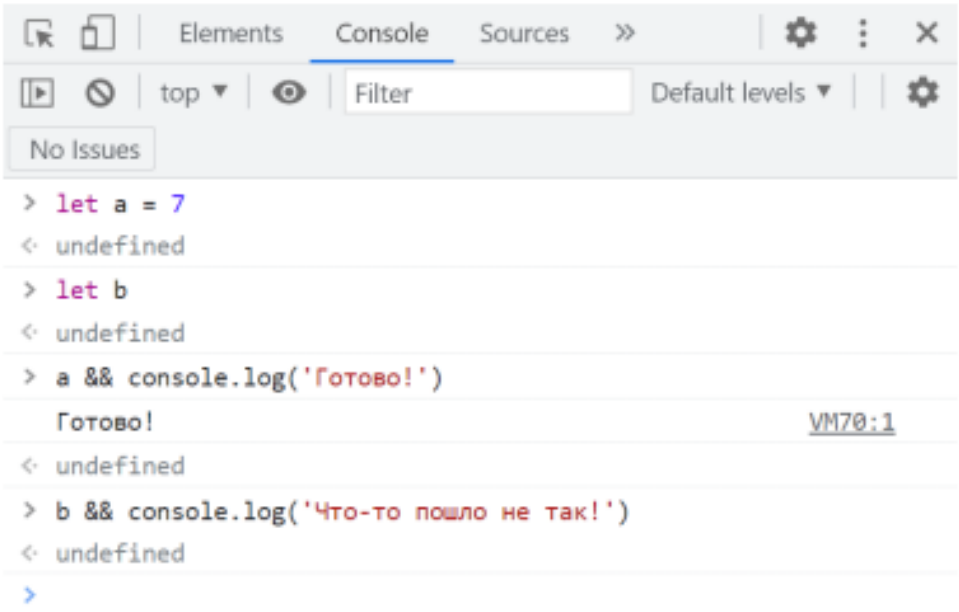
В противном случае, когда exprA приводится к истине, то возвращается результат выражения exprB, не зависимо от того истинно оно или ложно, т.к. данный операнд является последним.

```
'' && 'JavaScript' // '', т.к. первое выражение (пустая строка) приводится к false, то оно сразу и возвращается
3 === 3 && 4 // 4, т.к. первое выражение истинно, то будет возвращен результат второго выражения
'ab' + 'c' && !isNaN(5) // true, т.к. первое выражение истинно, то будет возвращен результат второго выражения
5 && 3 === 4 // false, т.к. 5 истинно, то возвращается результат вычисления второго выражения
'строка' && undefined // undefined, т.к. 'строка' приводится к true, то возвращается результат 2 операнда
'строка1' && 'строка2' // 'строка2', т.к. 'строка1' приводится к true, то возвращается результат 2 выражения
```



Пример вызова функций в зависимости от того, какие значения имеют те или другие переменные:

```
let a = 7;
let b;
a && console.log('Готово!');
b && console.log('Что-то пошло не так!')
```



В первом выражении `a && console.log('Готово!')` сначала будет найден результат выражения `a`. Он является `7` и не ложным, т.к. `Boolean(7)` это `true`. А так как результат не ложный, то будет вычисляться `console.log('Готово!')`. В итоге мы увидим в консоли сообщение «Готово!» и в качестве результате всего этого выражения значение `undefined`, т.к. данное значение возвращает этот метод.

Выполнение `b && console.log('Что-то пошло не так!')` также начинается с вычисления первого выражения, которое является `b`. Его результат `undefined`, а `undefined` приводится к `false`. А так как оно является ложным, то оно сразу возвращается как результат всего этого выражения. Второй операнд `console.log('Что-то пошло не так!')` не вычисляется и в консоли мы не увидим сообщение «Что-то пошло не так!».

Как вычисляется выражение с «||»

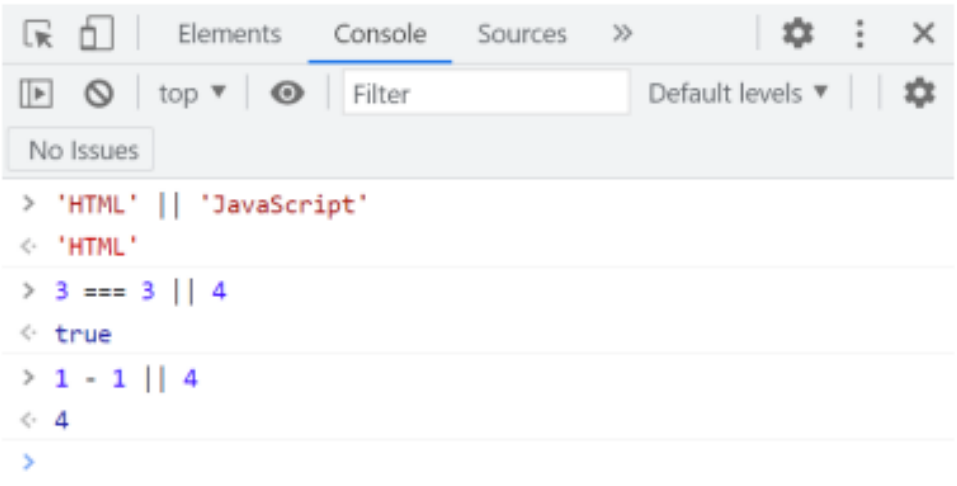
Пример выражения с оператором «логическое ИЛИ»:

```
exprA || exprB
```

Если выражение `exprA` приводится к `true`, то `exprB` не вычисляется и сразу же возвращается результат `exprA`.

В противном случае вернётся результат выражения `exprB`, т.к. он последний.

```
'HTML' || 'JavaScript' // 'HTML', т.к. первое выражение приводится к true
3 === 3 || 4 // true, т.к. первое выражение вычисляется как истинно
1 - 1 || 4 // 4, т.к. первое выражение ложно, следовательно будет возвращено второе выражение
```



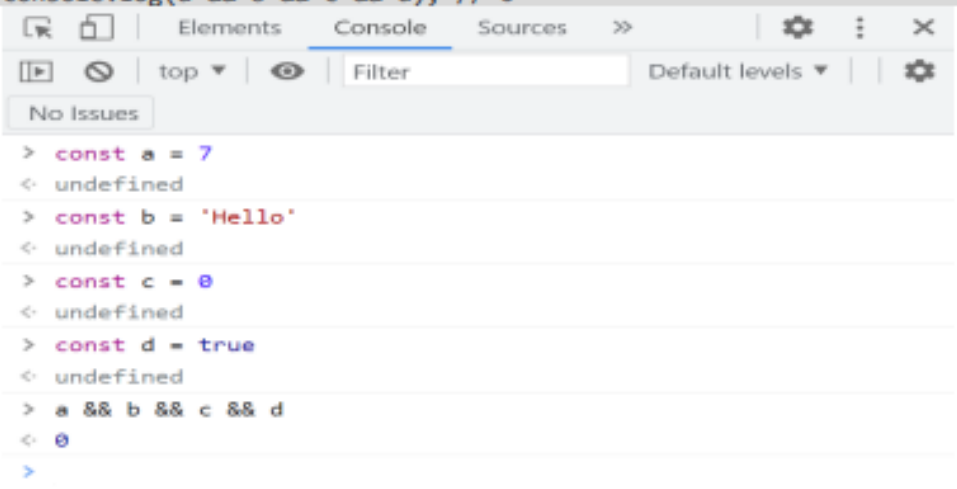
Оператор `||` очень часто применяется, когда мы хотим какой-то переменной присвоить дефолтное значение, если нет значения у другой переменной:

```
let varA;
const varB = varA || 0;
console.log(varB); // 0
```

Несколько операторов «&&» и «||»

Пример, содержащий несколько операторов `&&`:

```
const a = 7;
const b = 'Hello';
const c = 0;
const d = true;
console.log(a && b && c && d); // 0
```



Вычисление результата выражения, состоящего из цепочки операторов `&&` выполняется по точно такому же принципу. То есть мы ищем первое ложное значение и сразу же его возвращаем. После него другие выражения мы не вычисляем. Если все операнды являются истинными, то возвращается значение последнего операнда.

В этом коде `a` истинно, `b` истинно, `c` – нет. Значит возвращаем значение `c`, выражение `d` не вычисляем, т.к. результат уже найден.

Пример с несколькими операторами `||`:

```
const a = 0;
const b = '';
const c = 7;
const d = false;
console.log(a || b || c || d); // 7
```

В случае с `||` мы ищем первое истинное значение. Если не один из операндов не является истинным, то возвращаем результат вычисления последнего операнда.

В этом примере `a` ложно, `b` ложно, `c` – нет. Значит возвращаем значение `c`. Выражение `d` не вычисляем, т.к. результат уже найден.

Оператор ??

`??` - это ещё один логический оператор. Называется он оператором нулевого слияния (на английском *nullish coalescing operator*).

Этот оператор является бинарным:

```
exprA ?? exprB
```

Работает он очень просто: возвращает результат выражения `exprB`, если `exprA` вычисляется как `null` или `undefined`. В противном случае результат выражения `exprA`.

Оператор `??` очень похож на логический оператор «ИЛИ». Разница лишь в том, что оператор нулевого слияния рассматривает `null` и `undefined` как ложные значения, а все остальные – как истинные.

```
const a = '';
const b;
const c = null;
const d = 0;

const resultA = a ?? 'Привет, мир!'; // ""
const resultB = b ?? 'Привет, мир!'; // "Привет, мир!"
const resultC = c ?? 'Привет, мир!'; // "Привет, мир!"
const resultD = d ?? 'Привет, мир!'; // 0
```

В этом коде возвращается значение второго операнда только для примеров, в которых первый операнд равняется `null` или `undefined`. В остальных случаях значение первого операнда.

Оператор `??` также как `&&` и `||` не вычисляет следующие операнды, если результат уже известен:

```
const a = () => {
  console.log('Сообщение A');
```

```
  return false;
}
const b = () => {
  console.log('Сообщение B');
}

console.log(a() ?? b());
```

В этом выражении `a() ?? b()` у нас сначала вычисляется `a()`. В данном случае мы увидим в консоли сообщение «Сообщение A», а его результат будет `false`. А так как его результат не равно `null` или `undefined`, то оно будет сразу возвращено в качестве результата всего этого выражения. Таким образом выражение `b()` не будет вычисляться и мы не увидим в консоли сообщение «Сообщение B».

Побитовые операторы

Побитовые операторы предназначены для выполнения логических операций над целыми числами, а точнее над их двоичными представлениями.

Представление числа в двоичной системе осуществляется посредством 32 битов:

```
// 5  => 0000 0000 0000 0000 0000 0000 0000 0101
// 10 => 0000 0000 0000 0000 0000 0000 0000 1010
```

В JavaScript имеются следующие побитовые операторы:

- побитовое И `&`;
- побитовое ИЛИ `|`;
- побитовое НЕ `~`;
- исключающее ИЛИ `^`;
- сдвиг влево `<<`;

- сдвиг вправо `>>`;
- сдвиг вправо с заполнением нулями `>>>`.

Пример с `&`:

```
5 & 10; // 0
// 5      => ... 0000 0101
// 10     => ... 0000 1010
// 5 & 10 => ... 0000 0000
// 0 в двоичной (2) или 0 в десятичной (10)

5 & 7; // 5
// 5      => ... 0000 0101
// 7      => ... 0000 0111
// 5 & 7  => ... 0000 0101
// 101 в двоичной (2) или 5 в десятичной (10)
```

Выполнение логической операции «И» осуществляется для каждой пары битов, находящихся на одинаковых позициях в двоичных представлениях операндов. Логическое «И» работает так: `0 & 0 = 0`, `1 & 0 = 0`, `0 & 1 = 0` и `1 & 1 = 1`.

Пример с `|`:

```
5 | 10; // 15
// 5      => ... 0000 0101
// 10     => ... 0000 1010
// 5 | 10 => ... 0000 1111
// 1111(2) или 15(10)

5 | 4; // 5
// 5      => ... 0000 0101
// 4      => ... 0000 0100
// 5 | 4  => ... 0000 0101
// 101(2) или 5(10)
```

Выполнение логической операции «ИЛИ» осуществляется для каждой пары битов, находящихся на одинаковых позициях в двоичных представлениях операндов. Логическое «ИЛИ» работает так: `0 | 0 = 0`, `1 | 0 = 1`, `0 | 1 = 1` и `1 | 1 = 1`.

Примеры с `^`:

```
5 ^ 7; // 2
// 5      => ... 0000 0101
// 7      => ... 0000 0111
// 5 ^ 7  => ... 0000 0010
// 10(2) или 2(10)

5 ^ 4; // 1
// 5      => ... 0000 0101
// 4      => ... 0000 0100
// 5 ^ 4  => ... 0000 0001
// 1(2) или 1(10)
```

Выполнение логической операции «исключающее ИЛИ» осуществляется для каждой пары битов, находящихся на одинаковых позициях в двоичных

представлениях операндов. «Исключающее ИЛИ» работает так: `0 ^ 0 = 0`, `1 ^ 0 = 1`, `0 ^ 1 = 1` и `1 ^ 1 = 0`.

Пример с `~`:

```
~5; // -6
// 5       => 0000 0000 0000 0000 0000 0000 0000 0101
// ~5 (-6) => 1111 1111 1111 1111 1111 1111 1111 1010
```

Выполнение логической операции «НЕ» осуществляется для каждого бита двоичного представления операнда (число 0 заменяется на 1, а 1 на 0).

Пример с `<<`:

```
5 << 3; // 40
// 5      => 0000 0000 0000 0000 0000 0000 0000 0101
// 5 << 3 => 0000 0000 0000 0000 0000 0000 0010 1000
// 101000(2) или 40(10)
```

В `operand1 << operand2` оператор `<<` сдвигает двоичное представление `operand1` на количество битов, указанных посредством второго операнда `operand2`, добавляя нули справа.

Пример с `>>`:

```
45 >> 3; // 5
// 45      => 0000 0000 0000 0000 0000 0000 0010 1101
// 45 >> 3 => 0000 0000 0000 0000 0000 0000 0000 0101
// 101(2) или 5(10)
```

В `operand1 >> operand2` оператор `>>` сдвигает двоичное представление `operand1` на количество битов, указанных посредством второго операнда `operand2`. При этом лишние биты, сдвинутые вправо, отбрасываются.

Пример с `>>>`:

```
-30 >>> 3; // 536870908
// -30       => 1111 1111 1111 1111 1111 1111 1110 0010
// -30 >>> 3 => 0001 1111 1111 1111 1111 1111 1111 1100
// 11111111111111111111111111100(2) = 536870908(10)
```

В `operand1 >>> operand2` оператор `>>>` сдвигает двоичное представление `operand1` на количество битов, указанных посредством второго операнда `operand2`. При этом лишние биты, сдвинутые вправо, отбрасываются; число слева дополняется нулевыми битами. Для неотрицательных чисел сдвиг вправо с заполнением нулями и сдвиг вправо с переносом знака дают одинаковый результат.