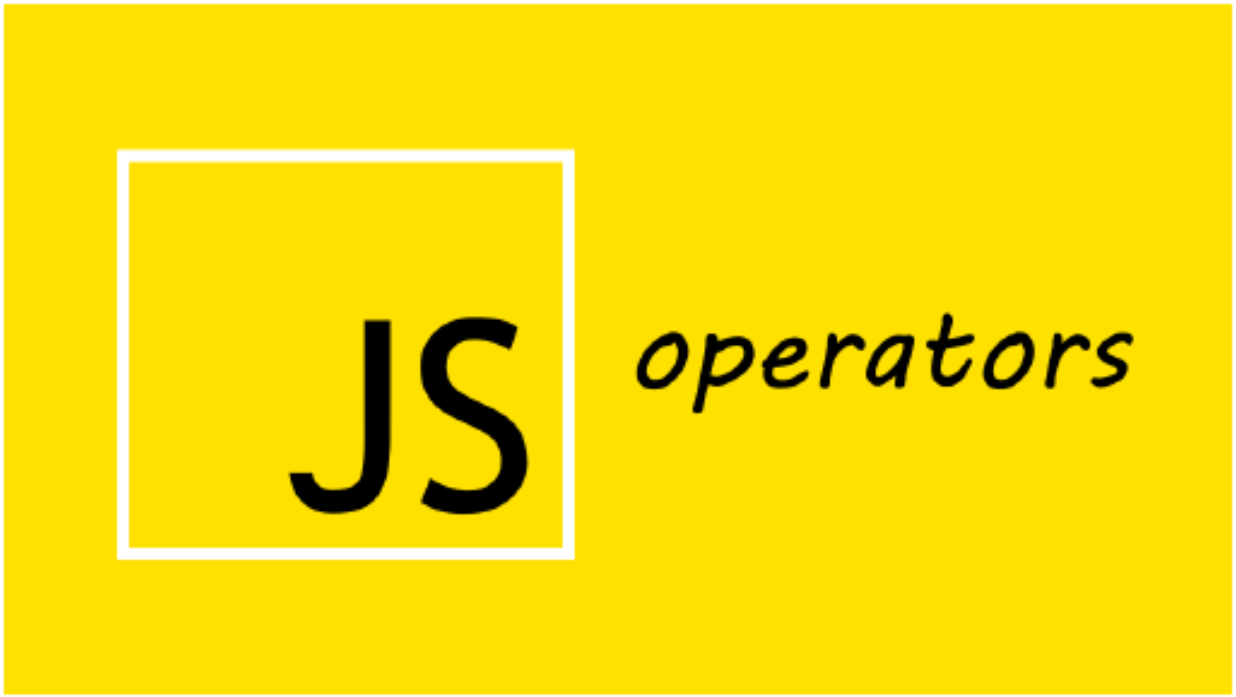
Лабораторная работа №3. Операторы в JavaScript



Содержание:

В JavaScript очень много операторов и их можно разделить на следующие группы:

- присваивания;
- сравнения;
- арифметические;
- побитовые;
- логические;
- строковые;
- условный (тернарный);
- запятая;
- текстовые.

Оператор – это просто внутренняя функция JavaScript. При использовании того или иного оператора мы, по сути, просто запускаем ту или иную встроенную функцию, которая выполняет какие-то определённые действия и возвращает результат.

Унарные и бинарные операторы

Каждый оператор оперирует **определённым количеством операндов**. Операнд – это то, что находится слева и (или) справа от оператора. Иногда операнды называют ещё аргументами.

Например, оператор присваивания используется с двумя операндами:

Операторы по количеству используемых операндов делятся на **унарные** и **бинарные**.

У **унарных операторов всегда один операнд (аргумент)**. Примеры унарных операторов:

```
delete obj.a
b++
typeof c
new Date()
+d
```

В этом примере:

- оператор `delete` удаляет свойство объекта `obj.a`;
- `++` увеличивает значение операнда `b` на `1`;
- `typeof` позволяет узнать тип операнда `c`;
- оператор `new` создаёт новый экземпляр объекта типа `Date`;
- `+` конвертирует значение операнда `d` к числу.

У **бинарных операторов** в отличие от унарных **всегда два операнда (аргумента)**. Примеры бинарных операторов:

```
a += 7
c - b
d = 'Привет'
e === 4
f1 && f2
```

Здесь:

- оператор `+=` имеет два операнда `a` и число `7`; он складывает значение переменной `a` с числом `7`, а затем полученный результат присваивает `a`;
- `-` отнимает от значения операнда `c` значение операнда `b`;
- `=` присваивает `'Привет'` переменной `d`;
- `===` сравнивает значения `e` и `4` с учетом типа;
- `&&` вычисляет логическое «И» операндов `f1` и `f2`.

Нотации записей операторов

В зависимости от того, в каком стиле записан оператор относительно операнда(ов) он имеет инфиксную, префиксную или постфиксную запись.

Инфиксную форму записи имеют операторы, которые находятся между операндами. Т.е. это все бинарные операторы:

```
10 / 2
'PI' in Math
myObj instanceof Object
bar = 2
x += y
c && d
```

Префиксную запись имеют операторы, которые пишутся перед операндом:

```
delete Employee.age
++num
+true
typeof Infinity
```

Постфиксную нотацию имеют операторы, которые идут после операнда:

```
num--
i++
myFunc()
```

Круглые скобки это тоже оператор в JavaScript. В данном случае он сообщает интерпретатору JavaScript, что необходимо вызвать функцию `myFunc`.

Некоторые операторы в JavaScript, как например, `++` и `--` имеют как префиксную, так и постфиксную запись.

Операторы присваивания

Оператор присваивания обозначается с помощью знака `=`. Он используется, когда вам нужно присвоить той или иной переменной какое-то значение.

```
let a;
a = 7;
```

В данном примере на второй строчке мы переменной `a`, которая была объявлена ранее, присваиваем значение `7`.

Что же такое оператор, например, присваивания или любой другой?

Оператор присваивания или любой другой можно представить себе как некоторую встроенную функцию в JavaScript в данном случае с именем `=`, которая выполняет некоторые действия и возвращает какой-то результат.

Как мы знаем у операторов имеются операнды, в данном случае их два. Один расположен слева от `=`, а другой справа. Эти операнды являются аргументами, которые мы передаём в эту функцию `=`.

После получения этих аргументов, в данном случае переменной `a` и выражения `7`, функция выполняет следующие действия:

- вычисляет результат выражения `7`;
- пытается найти переменную `a`; если переменная не найдена выбрасывает ошибку;
- присваивает переменной `a` результат выражения `7`;
- возвращает в качестве результата выполнения функции `=` результат выражения `7`.

Это краткий пример того, как внутри JavaScript работает оператор `=`.

```
const obj = {};
const arr = ['a', 'b', 'c'];
let x;
let y = 5;
// присвоим свойству a объекта obj значение 'abc'
obj.a = 'abc';
// присвоим 2 элементу массива значение 'z'
arr[1] = 'z';
// присвоим переменной x значение переменной y
x = y;
```

В JavaScript имеются ещё составные операторы присваивания: `+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`.

Данные операторы перед присваиванием, выполняют ещё одно дополнительное действие. Например, оператор `+=` складывает значение переменной с результатом выражения, указанного справа от `+=`:

```
let x = 7;
x += 5; // 12
```

То есть инструкцию `x =+ 5` по другому можно записать так:

```
let x = 7;
x = x + 5; // 12
```

По сути, `x += 5` это просто сокращенная запись от `x = x + 5`.

Ещё примеры:

```
let x = 7;
x += 8; // x = x + 8
x **= 3; // x = x ** 3
x -= 30;  // x = x - 30
```

```
x *= 2;  // x = x * 2
```

Арифметические операторы

В JavaScript выделяют следующие **математические операторы**:

- унарный плюс `+` и минус `-` ;
- сложение `+`;
- вычитание `-`;
- умножение `*`;
- деление `/`;
- остаток от деления `%`;
- возведение в степень `**`;
- увеличение значения переменной на 1 `++`;
- уменьшение значения переменной на 1 `--`.

```
console.log(8 + 4);  // 12
console.log(8 - 4);  // 4
console.log(8 * 4);  // 32
console.log(8 / 4);  // 2
console.log(8 % 3);  // 2  (остатка от деления, т.е. 8 - 3 * 2 = 2)
console.log(2 ** 3); // 8  (возведение числа 2 в степень 3)
let num = 7;
console.log(++num); // 8
console.log(--num); // 7
```

В результате **выполнения математической операции** всегда возвращается **числовое значение**.

```
5 - 15;      // -10 (1 пример)
'5' - '15';  // -10 (2 пример)
```

JavaScript – это язык с динамическим приведением типов. Во втором примере оператор **минус** (–) является **математическим** и может оперировать **только с числами**. Для выполнения этой операции JavaScript **приведёт значения первого и второго операнда к числам** (5 и 15), а затем уже выполнит **вычитание**.

Ещё один пример с использованием оператора **минус** (`-`):

```
5 - 'px'; // NaN
```

В этом примере JavaScript приведёт строку **«px»** к специальному числовому значению **NaN**. В результате вычисления **5 – NaN** будет возвращён результат **NaN**.

Пример с использованием оператора **плюс** (`+`):

```
5 + 'px';  // "5px"
```

Оператор плюс (+) не только **математический**, он также используется **для конкатенации (соединения) строк**.

Выбор операции при использовании оператора **плюс** (`+`) зависит от **типа значений операндов**. Если хотя бы один из операндов является **строкой**, то данная операция

рассматривается как операция **конкатенации строк**. В противном случае она будет считаться **математической**.

В этом примере **второй операнд является строкой**, следовательно, данная операция будет операцией **конкатенацией строк**. Т.к. первый операндов не является строкой, то JavaScript приведёт его к ней. В результате будет возвращена строка «5px».

Ещё оператор **плюс** (+) может применяться **к одному операнду**:

```
+5.8;      // 5.8 (при применении к числу унарный оператор + ничего не изменяет)
+'5.8';     // 5.8 (при применении унарного оператора + к не числовому значению приводит его к числу)
+'5.8px';  // NaN (привёл текст к числовому значению NaN)
```

Унарный оператор **плюс** (+) в основном используется для **явного преобразования не числовых типов данных к числам**.

Оператор **минус** (–), также как и оператор **плюс** (+) может применяться **к одному операнду**. Если операнд является числом, то он просто **изменит его знак на противоположный**. В противном случае, он преобразует значение операнда к числу, а затем изменит его знак на противоположный.

```
5 + (-10); //-5;
-(-15);    // 15
-'5';      // -5
-'5px';    // NaN
```

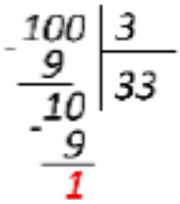
Операторы **инкремента** (++) и **декремента** (--) имеют префиксную и постфиксную форму.

```
var num1, num2;
num1 = 5;
num2 = num1++; // постфиксная форма, т.е. сначала возвращается значение, а затем увеличивается значение переменной num1 на 1.
console.log(num1); // 6
console.log(num2); // 5
num1 = 5;
num2 = ++num1; // префиксная форма, т.е. сначала увеличивается значение переменной num1 на 1, а затем возвращается её значение
console.log(num1); // 6
console.log(num2); // 6
```

Оператор **процент** (%) вычисляет остаток от деления первого операнда на второй:

```
100 % 3; // 1
```

Как вычисляется остаток от деления показано на картинке:



Операторы сравнения

Операторы сравнения предназначены для сравнения двух операндов. Если сравнение верное, то в качестве результата этой операции возвращается `true`. В противном случае, значение `false`.

В JavaScript выделяют следующие операторы сравнения: `==` (равенство), `!=` (не равенство), `===` (строгое равенство), `!==` (строгое не равенство), `>` (больше), `>=` (больше или равно), `<` (меньше), `<=` (меньше или равно).

```
// СРАВНЕНИЕ ЧИСЕЛ
5 > 10;         // false
5 === 2 + 3;    // true
7 < 2 * 3;      // false
4 <= 4;         // true
+0 === -0;      // true
// СРАВНЕНИЕ БУЛЕВЫХ ЗНАЧЕНИЙ
true > false;  // true
true == false; // false
// РАВЕНСТВО null и undefined
null == undefined; // true (особенность языка)
```

Строки сравниваются посимвольно. Т.е. сначала первые символы операндов. Если первый символ первого операнда большое второго, то значит первый операнд больше второго. Если первые символы равны, то сравнивается вторые символы операндов и т.д.

При этом какой символ больше другого определяется по их кодам в таблице Unicode.

```
// ПРИМЕР 1
// 51 - код символа '3' // '3'.charCodeAt(0)
// 50 - код символа '2' // '2'.charCodeAt(0)
'3' > '25'; // true, т.к. 51 > 50

// ПРИМЕР 2
// 1080 - код символа 'и' // 'и'.charCodeAt(0)
// 1090 - код символа 'т' // 'т'.charCodeAt(0)
'символ' > 'строка'; // false, т.к. 1080 не больше 1090

// ПРИМЕР 3
'домик' > 'дом'; // true, т.к. у него больше символов
```

При сравнении маленькие и большие буквы алфавита не равны, т.к. имеют разные коды в таблице Unicode. Поэтому при операциях сравнения строки желательно приводить к одному регистру.

```
// 1044 - код символа 'Д' // 'Д'.charCodeAt(0)
// 1076 - код символа 'д' // 'д'.charCodeAt(0)
'Дом' === 'дом'; // false, т.к. 'Д' не равно 'д'

'Дом'.toLowerCase() === 'дом'.toLowerCase(); // true
```

Если **оба операнда являются объектами**, то сравниваются их ссылки.

```
var
  obj1 = {
    num: 3
  },
  obj2 = {
    num: 3
  },
  obj3 = obj1;

obj1 == obj2;   // false, т.к. переменные имеют разные ссылки
obj1 == obj3;   // true, т.к. переменные ссылаются на один и тот же объект
obj1 === obj3;  // true, т.к. переменные ссылаются на один и тот же объект
```

Во всех операциях, кроме операций с использованием строго равенства и строго не равенства, JavaScript перед операцией сравнения **приводит их к одному типу**. Данное действие он конечно же выполняет только в том случае, если **операнды имеют разный тип данных**.

Если операнды имеют **примитивный тип данных**, то они приводится при операции сравнения **к числу**:

```
'2' == 2;    // true, т.к. строка '2' будет приведена к числу 2
true != 0;   // true, т.к. true будет приведено к 1
false == 0;  // true, т.к. false == true
'' > -5;     // true, т.к. пустая строка будет приведена к 0
'' == false; // true, т.к. пустая строка будет приведена к 0 и false тоже к 0
'5px' == 5;  // false, т.к. строка '5px' будет приведена к NaN
```

При сравнении объекта с примитивным типом данных, JavaScript вызывает его метод `valueOf` или `toString`. Полученное значение, если оно не объект, сравнивается с примитивным типом данных и возвращается результат. Если метод `valueOf` или `toString`, то выбрасывается ошибка «Cannot convert object to primitive value».

```
// ПРИМЕР 1
var obj = {
  num: 5,
  toString: function () {
    return this.num;
  }
};
obj == 5; // true

// ПРИМЕР 2
var
  arr = [4, 5, 6],
  str = '4,5,6';
arr == str; // true
```

В большинстве случаев при выполнении операций равенства или не равенства, их желательно выполнять с использованием операторов `===` и `!==`. Применение данных операторов позволит исключить неожиданные ситуации при сравнении операндов с различным типом данных.

```
2 === '2';  // false, т.к. тип данных разный
true === 1; // false, т.к. тип данных разный
```

Оператор == сравнивает на равенство, а вот === — на идентичность. Плюс оператора === состоит в том, что он не приводит два значения к одному типу. Именно из-за этого он обычно и используется.

```
abc  == undefined; // true, если abc = undefined | null
```

```
abc === undefined;       // true - только если abc = undefined!
```

```
abc  == false;   // true, если abc = false | 0 | '' | []
abc === false; // true, только если abc = false!
```

Строковые операторы

В JavaScript оператор `+` можно использовать также для конкатенации или, другими словами, объединения строк:

```
'Hello, ' + 'world!' // "Hello, world!"
```

В этом примере оператор`+` соединяет две строки. В результате вы получите `"Hello, world!"`.

В качестве операндов вы можете использовать переменные:

```
const strA = 'Hello,';
const strB = 'world!';
const strC = strA + ' ' + strB;
```

В JavaScript создавать строковые значения можно также с помощью **шаблонной строки** (template string literal). В отличие от строк, задаваемых с помощью одинарных или двойных кавычек, этот механизм имеет множество мощных возможностей.

Одна из возможностей – это использовать JavaScript выражения внутри строк. Пример, приведённый выше, с использованием шаблонной строки:

```
const strA = 'Hello,';
const strB = 'world!';
const strC = `${strA} ${strB}`;
```

Обратите внимание, что шаблонная строка заключается в обратные кавычки. Внутри неё можно помещать различные выражения JavaScript, заключая их в `${}`.

Здесь между выражениями стоит пробел. Он необходим, чтобы у нас получилась точно такая же строка как в предыдущем примере.

Шаблонные строки можно очень просто использовать для создания многострочных строк:

```
const multipleLines = `Эта строка имеет
```

```
несколько строчек`;
```
Ещё примеры с использованием шаблонной строки:

```
const sum = (a, b) => a + b;
const a = 7;
const b = 3;
const result = `Сумма ${a} и ${b} равно ${sum(a, b)}.`
console.log(result);
```
Пример соединения числа со строкой:

```
'Bob' + 50 // "Bob50"
```
В данном случае интерпретатор JavaScript автоматически конвертировал число `50` в строку `"50"`, а потом соединил две строки: `"Bob"` и `"50"`. Данный процесс в JavaScript называется **автоматическим приведением типов**.

Что избежать таких ситуаций, т.е. не получить в результате то, чего вы не ожидали необходимо чётко знать типы значений, которые вы хотите соединить. То есть если в качестве результата вы хотите получить строку, то нужно чтобы все соединяемые значение были строками.

Текстовые операторы

Текстовые операторы – это такие, которые записываются не с помощью значков, а посредством символов. Например:

- `typeof` – позволяет узнать тип того или иного значения;
- `instanceOf` – можно проверить принадлежность объекта к тому или иному классу;
- `delete` – удаляет то или иное свойство объекта.

Приоритет операторов

Операторов в JavaScript как вы уже знаете очень много. У всех них имеется определённый **приоритет**. Без него не обойтись, когда выражение строится с использованием нескольких операторов. В этом случае их приоритет будет определять порядок, в соответствии с которым они выполняются. Операторы, имеющие более высокий приоритет выполняются первыми:

```
3 - 5 * 4 + 2 // -15
```
В этом примере оператор `*` имеет более высокий приоритет. Поэтому сначала выполнится умножение, а потом вычитание и сложение.

Но с использованием оператора «группировка» `( ... )` порядок выполнения выражение будет другим, т.к. он имеет более высокий приоритет чем `*`:

```
(3 - 5) * (4 + 2) // -12
```
Список операторов, упорядоченных по приоритету (от более высокого к низкому, через тире указана их специфичность):

1. Группировка `()` – не определено;
2. Доступ к свойствам `.`, `[]`, вызов функции `()`, `?.` – слева направо, создание экземпляра объекта со списком аргументов `new` – не определено;
3. создание экземпляра объекта без аргументов `new` – справа налево;
4. Постфиксный инкремент `++` и декремент `--` – не определено;
5. Логическое `!` и побитовое отрицание `~`, унарный плюс `+` и минус `-`, префиксный инкремент `++` и декремент `--`, `typeof`, `void`, `delete`, `await` – справа налево;
6. Возведение в степень `**` – справа налево;
7. `*`, `/`, `%` – слева направо;
8. Сложение `+`, вычитание `-` – слева направо;
9. `<<`, `>>` и `>>>` – слева направо;
10. `<`, `<=`, `>` и `>=`, `in` и `instanceof` – слева направо;
11. `==`, `!=`, `===` и `!==` – слева направо;
12. `&` – слева направо;
13. `^` – слева направо;
14. `|` – слева направо;
15. `&&` – слева направо;
16. `||` и `??` – слева направо;
17. Условный тернарный оператор `?:` – справа налево;
18. `=`, `+=`, `-=`, `**=`, `*=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `>>>=`, `&=`, `^=`, `|=`, `&&=`, `||=` и `??=` – слева направо, `yield` и `yield*` – справа налево;
19. Запятая `,` – слева направо.

Пример выражения:

```
20 * 3 % 2 // 0
```

В этом выражении операторы `*` и `%` имеют одинаковый приоритет, поэтому порядок выполнения этого выражения будет определяться в соответствии с ассоциативностью. Т.к. оператор `*` имеет левую ассоциативность (слева направо), то сначала будет выполнено `20 * 3`, а потом остаток от деления. Другими словами, это выражение будет обрабатываться так:

```
(20 * 3) % 2 // 0
```

Если бы оператор `*` имел бы правую ассоциативность, то это выражение интерпретировалась бы так:

```
20 * (3 % 2) // 20
```

Например, оператор присваивания имеет правую ассоциативность (справа налево):

```
let a, b;
a = b = 7;
```

Т.е. в этом выражении сначала выполнится `b = 7`, а затем результат этого выражения будет присвоен переменной `a`. Т.е. так:

```
a = (b = 7);
```