Основные операторы JavaScript

Операторы и их приоритеты выполнения. Условные операторы и циклы.

[Введение](#_3znysh7)

[Операторы в JavaScript](#_tyjcwt)

[Принципы ветвления, визуализация, блок-схемы](#_3dy6vkm)

[Операторы if, if-else](#_1t3h5sf)

[Оператор switch](#_2s8eyo1)

[Тернарный оператор](#_17dp8vu)

[Комбинации условий](#_3rdcrjn)

[Функции](#_26in1rg)

[Области видимости](#_lnxbz9)

[Рекурсия](#_35nkun2)

[Практикум. Угадай число](#_1ksv4uv)

Практическое [задание](#_44sinio)

[Дополнительные материалы](#_p6o0l6g458ly)

[Используемая литература](#_o592u646fkiw)

# Введение

Вы уже знаете, что собой представляют переменные в JavaScript, каких они бывают типов, как они применяются в выражениях. Этих знаний вполне хватит, чтобы написать простую, работающую полезную программу. Но функционал языка гораздо шире.

Критерий истины — практика, поэтому новые знания будем усваивать через реализацию игр.

# Операторы в JavaScript

Как и в любом языке программирования, в JavaScript есть операторы. Сам по себе оператор — это наименьшая автономная часть языка программирования, то есть команда. У операторов есть операнды, или аргументы оператора — сущности, к которым применяется оператор. При сложении двух чисел (3+2) работает оператор сложения с двумя операндами.

Операторы бывают унарными и бинарными. Унарный оператор применяется к одному операнду:

|  |
| --- |
| var x = 1; x = -x; *// унарный минус* |

Бинарный — к двум:

|  |
| --- |
| var a = 1; var b = 2; a + b; *// бинарный плюс* |

У некоторых операторов есть особые названия:

* **инкремент** — означает увеличение операнда на установленный фиксированный шаг (как правило, единицу). Он же **a++** или **a+1**;
* **декремент** — обратная инкременту операция: **a--** или **a-1**;
* **конкатенация** — сложение строк. Обратной операции нет.

|  |
| --- |
| var a = "моя" + "строка"; |

При выполнении бинарных операторов нужно помнить, что JavaScript будет преобразовывать типы операндов, если они различаются.

При конкатенации, если в операторе один из операндов — строка, то и остальные операнды будут преобразованы к строке вне зависимости от их порядка.

|  |
| --- |
| alert("1" + 2 ); *// "12"* alert( 2 + "1" ); *// "21"* |

При выполнении других арифметических операторов такого приведения типов не будет — все операнды будут приводиться к числу.

|  |
| --- |
| alert("2" - 1 ); *// 1* alert( 9 / "3" ); *// 3* |

Чтобы работать со сложными выражениями, содержащими более одного оператора, надо определять приоритеты операций, порядок их выполнения.

Если с арифметическими операторами все просто — работает классическая логика (например, сначала умножение, потом сложение), то с программными операторами JavaScript сложнее. Их приоритеты упорядочены в таблице в порядке убывания важности:

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| . [ ] ( ) | Доступ к полям, индексация массивов, вызовы функций и группировка выражений |
| ++ -- - ~ ! delete new typeof void | Унарные операторы, тип возвращаемых данных, создание объектов, неопределенные значения |
| \* / % | Умножение, деление, деление по модулю |
| + - + | Сложение, вычитание, объединение строк |
| << >> >>> | Сдвиг бит |
| < <= > >= instanceof | Меньше, меньше или равно, больше, больше или равно, instanceof |
| == != === !== | Равенство, неравенство, строгое равенство, строгое неравенство |
| & | Побитовое И |
| ^ | Побитовое исключающее ИЛИ |
| | | Побитовое ИЛИ |
| && | Логическое И |
| || | Логическое ИЛИ |
| ?: | Условный оператор |
| = *OP*= | Присваивание, присваивание с операцией (например, += и &=) |
| , | Вычисление нескольких выражений |

Согласно таблице, при выполнении этого выражения сначала рассчитывается арифметическая часть, а потом происходит присвоение, так как оно находится ниже, чем сложение и умножение:

|  |
| --- |
| var a = 5 \* 3 - 7; |

Обратим внимание на унарные операторы инкрементрирования и декрементирования. В JavaScript есть префиксная и постфиксная форма их записи. По сути, обе увеличивают значение операнда на единицу. Но посмотрим, как они это делают:

|  |
| --- |
| var a = 5; alert(a++); *// выведет 5* alert(++a); *// выведет 7* |

В постфиксной форме сначала происходит возвращение значения, а потом выполняется инкрементирование. В префиксной форме инкрементирование производится сразу, а возврат — уже с обновленным значением.

В JS есть и операторы сравнения, которые возвращают логическое значение:

|  |
| --- |
| alert( 2 > 1 ); *// true* alert( 2 >= 1 ); *// true* alert( 2 == 1 ); *// false* alert( 2 != 1 ); *// true* alert("Б" > "А" ); *// true* |

При сравнении строк из нескольких букв операция выполняется пошагово: сначала сравниваются первые буквы, потом вторые и так далее.

Не стоит забывать и о числовом преобразовании:

|  |
| --- |
| alert("2" > 1 ); *// true* alert("01" == 1 ); *// true* alert( false == 0 ); *// true, значение false становится числом 0* alert( true == 1 ); *// true, так как true становится числом 1.* alert( "" == false ); |

Для строгого сравнения на равенство применяется другой оператор:

|  |
| --- |
| alert( 0 === false ); *// false, т.к. типы различны* alert( 0 !== false ); *// true, т.к. типы различны* |

Значения **null** и **undefined** равны друг другу, но не чему бы то ни было еще. Это жесткое правило прописано в спецификации языка. При явном преобразовании в число (то есть вызванном пользователем) **null** принимает значение 0, а **undefined** — **NaN**.

# Принципы ветвления, визуализация, блок-схемы

В программном коде, как и в жизни, множество решений зависит от внешних факторов: «Если случится событие А, то я выполню действие Б». Именно по такому принципу строится ветвление во всех языках программирования.

Для ветвления в программировании применяются специальные операторы, обеспечивающие выполнение команды или набора команд только при условии истинности логического выражения или их группы. Ветвление — одна из трех базовых конструкций структурного программирования, наряду с последовательным выполнением команд и циклом.

*Для справки*: в дискретной математике (фундаментальной науке, лежащей в основе программирования) условие ветвления — это предикат. Почитать об этом можно по ссылке в разделе «Дополнительная литература».

Прежде чем приступать к написанию ветвлений на JavaScript, поговорим о случаях, когда на них влияет множество факторов. Тогда стоит визуализировать логику программы или ее части в виде блок-схемы, чтобы не запутаться при реализации.

Блок-схема — распространенный тип схем, описывающих алгоритмы или процессы, в которых отдельные шаги изображаются в виде блоков различной формы, соединенных линиями, которые указывают направление последовательности. Сама блок-схема состоит из стандартных элементов:

1. **Процесс** (функция обработки данных любого вида).
2. **Данные.**



1. **Предопределенный процесс** — символ отображает предопределенный процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте.
2. **Решение** — ситуация, имеющая одну точку входа и ряд альтернативных выходов, только один из которых можно использовать после вычисления условий, определенных внутри символа.





1. **Терминатор** (начало или конец программы)



Для наших нынешних целей перечисленных блоков вполне достаточно, а более подробный материал по блок-схемам можно найти по ссылке в «Дополнительной литературе».

Чтобы изучать ветвления, нам потребуется элемент «Решение».

## Операторы if, if-else

Для реализации ветвления в JS используется оператор **if**:

|  |
| --- |
| if( Условие ) {  Действие; } |





Условие — это любое выражение, возвращающее булевское значение (**true**, **false**), то есть вопрос, на который ответить можно только «да» или «нет». Если выражение возвращает значение, отличное от типа **boolean**, то оно будет автоматически к нему приведено. **0**, **null undefined**, **“”** и **NaN** будут транслированы в **false**, остальные значения — в **true**. Действие выполняется, когда условие истинно (**true**). Обычно условием является одна или несколько операций сравнения, объединенных логическими связками (И, ИЛИ). В результате проверки условия может выполняться сразу несколько операторов:

|  |
| --- |
| if( Условие ) {  Действие1;  Действие2; } |

Разберем вариант, когда одного условия недостаточно. Рассмотрим пример ветвления, когда в случае истины выполним одно действие, а иначе — другое.





|  |
| --- |
| if( Условие ) {  Действие1; } else{  Действие2; } |

Реализуем простой пример:

|  |
| --- |
| var x = 5; var y = 42; if( x > y )   alert (x + y); else  alert(x \* y); |

Если по условию нужно выполнять всего один оператор, можно не ставить фигурные скобки.

Но не всегда можно уложить логику ветвления в две ветки. JS позволяет разделять программу на сколько угодно вариантов с помощью конструкции **else if**, которая позволяет анализировать дополнительное условие. При этом выполняться будет первое условие, вернувшее **true**.

Представим следующую задачу: даны два произвольных числа, необходимо вывести на экран их соотношение друг с другом. По сути, здесь три варианта: либо первое число больше, либо второе, либо они равны.

|  |
| --- |
| var x = 5; var y = 42; if(x > y)   alert("x больше y"); else if ( x < y )  alert("x меньше y");  else  alert("x равно y"); |

## Оператор switch

Допустим, нужно разделить программу не на два или три варианта, а больше. Если много раз использовать конструкцию **else if**, это может серьезно ухудшить читаемость кода. Поэтому существует специальный оператор выбора из нескольких вариантов — **switch**. Его синтаксис:

|  |
| --- |
| switch(переменная){  case Значение1:  Действие1;  break;  case Значение2:  Действие2;  break;  default:  Действие3; } |

Оператор **switch** смотрит на значение переменной (или выражения, или возвращающего значения) и сравнивает его с предложенными вариантами. В случае совпадения выполняется соответствующий блок кода. Если по всем вариантам совпадения так и не обнаружилось, выполняются операторы из блока **default**. Он необязательный и может отсутствовать.

Обратите внимание на ключевое слово **break** в конце каждого блока **case**. Оно ставится в 99 % случаев и означает, что нужно прекратить выполнение операций внутри **switch**. Если в конце блока **case** нет оператора **break**, интерпретатор продолжает выполнять действия из следующих блоков.

|  |  |
| --- | --- |
| var now = 'evening'; switch (now){  case 'night':  alert('Доброй ночи!');  break; case 'morning':  alert('Доброе утро!');  break; case 'evening':  alert('Добрый вечер!');  break; default:  alert('Добрый день!');  break; } | var now = 'evening'; if (now == 'night'){  alert('Доброй ночи!'); } else if (now == 'morning'){  alert('Доброе утро!'); } else if (now == 'evening'){  alert('Добрый вечер!'); } else{  alert('Добрый день!'); } |

## Тернарный оператор

Тернарный оператор — это операция, возвращающая либо второй, либо третий операнд в зависимости от условия (первого операнда):

|  |
| --- |
| (Условие) ? (Оператор по истине) : (Оператор по лжи); |

Чтобы сохранить максимальное из двух произвольных чисел в переменную, вместо громоздких строк ветвления можно написать:

|  |
| --- |
| var x = 10; var y = 15; var max = (x > y) ? x : y; alert(max); |

Тернарный оператор — красивая возможность сделать код лаконичнее. Но, как и любым инструментом, им не стоит злоупотреблять.

По своей сути тернарный оператор отличается от **if**. Его нельзя использовать многократно, как в случае **if** и **else if**, — это засоряет код. Тернарный оператор нужен, чтобы встраивать небольшие условные ветки прямо в выражение — то есть он не заменяет стандартный **if-else**. Если надо описать условия непосредственно в выражении, следует использовать тернарный оператор. Но чтобы создать более сложное условие с телом, состоящим из нескольких инструкций, применяют **if** и **else**.

## Комбинации условий

В условном операторе можно комбинировать условия при помощи логических операций:

* **ИЛИ (x || y)** — если хотя бы один из аргументов **true**, то возвращает **true**, иначе — **false**;
* **И (x && y)** — возвращает **true**, если оба аргумента истинны, а иначе –— **false**;
* **НЕ (!x)** — возвращает противоположное значение.

|  |
| --- |
| alert( true || true ); *// true* alert( false || true ); *// true* alert( true || false ); *// true* alert( false || false ); *// false* alert( true && true ); *// true* alert( false && true ); *// false* alert( true && false ); *// false* alert( false && false ); *// false* alert( !true ); *// false* alert( !0 ); *// true* |

# Функции

Используя код одного из примеров, надо построить с пользователем диалог. Код в любой программе работает последовательно, строка за строкой. Значит условие уже отработано, вернуться к нему невозможно. Как решить эту задачу?

Можем скопировать весь блок операций еще несколько раз — но сколько именно? Да и copy-paste — совсем уж плохое решение. Будем использовать функции.

Функция — это блок кода, к которому можно обращаться из разных частей скрипта. Функции могут иметь входные и выходные параметры. Входные могут использоваться в операциях, которые содержит функция. Выходные устанавливаются функцией, а их значения используются после ее выполнения. Программист может создавать необходимые ему функции и логику их выполнения.

Если проводить аналогию с реальной жизнью, то функция — это навык скрипта. Ведь мы не учимся ходить каждый раз, когда перемещаемся — просто выполняем функцию «ходить». Так же и скрипт может иметь функцию **go**, которая может вызываться в любое время.

Функция в JS объявляется с помощью ключевого слова **function**. За ним следует ее название, которое мы придумываем сами. Затем в круглых скобках через запятую указываются параметры, которые данная функция принимает. По сути, параметры — это входные данные для функции, над которыми она будет выполнять какую-то работу. После параметров в фигурных скобках следует тело функции. Когда функция объявлена, можем ее вызвать и посмотреть, как она работает. Описание функции может находиться и до, и после ее вызова.

|  |
| --- |
| function имя\_функции(параметр1, параметр2, ...){  Действия } |

Создадим функцию, которая будет сравнивать числа:

|  |
| --- |
| function compare\_numbers(x, y){  if (x > y)  alert("x > y");  else if (x < y)  alert("x < y");  else  alert("x = y"); } compare\_numbers(10, 20); compare\_numbers(20, 10); compare\_numbers(20, 20); |

При вызове функции в нее нужно передавать такое количество параметров, которое заявили при ее создании. Их может быть от нуля и более. Если параметры не переданы, при вызове функции нужно просто указать пустые скобки.

Оператор **return** позволяет завершить выполнение функции, вернув конкретное значение. Если в функции не указано, что она возвращает, то результатом ее работы может быть только вывод текста на экран (см. предыдущую функцию). Но в большинстве случаев результат работы функции используется в программе. Тогда необходим оператор **return**. Напишем функцию, возвращающую среднее арифметическое двух чисел:

|  |
| --- |
| function average(x, y) {  return (x + y)/2; } avg = average(42, 100500); alert(avg); |

Так мы можем не только научить скрипт определенным навыкам, но и сохранить результат выполнения каждой функции для дальнейшего использования.

## Области видимости

При работе с функциями в JS нужно также помнить об областях видимости. Они бывают глобальными и локальными. Глобальными называют переменные и функции, которые не находятся внутри функции.

В JS все глобальные переменные и функции являются свойствами специального «глобального объекта» (global object). В браузере он явно доступен под именем **window**. Объект **window** одновременно является глобальным объектом и содержит ряд свойств и методов для работы с окном браузера.

Локальные переменные доступны только внутри функции. Если на момент определения функции переменная существовала, то она будет существовать и внутри функции, откуда бы ее ни вызывали.

|  |
| --- |
| function changeX(x){  x += 5;  alert(x); } var x = 1; alert(x); *// выводит 1* changeX(x); *// выводит 6* alert(x); *// выводит 1* |

## Рекурсия

Рекурсия — это вызов функцией самой себя, и это может быть полезно. Не будем сейчас углубляться в решение задач по обходу деревьев — приведем пример с вычислением последовательности **n** чисел Фибоначчи (последующее число равно сумме двух предыдущих). Каждый раз мы не знаем, сколько чисел Фибоначчи запросит пользователь, но, используя рекурсию, можем не думать об этом.

|  |
| --- |
| function fibonacci(n, prev1, prev2){  var current = prev1 + prev2;  var fibonacci\_string = current + " ";  if(n > 1)  fibonacci\_string += fibonacci(n - 1, current, prev1);  return fibonacci\_string; } alert(fibonacci(15, 1, 0)); |

Рекурсия важна для структур, у которых нет фиксированного количества уровней вложенности, но на каждом есть жесткая схема. Мы не можем сказать, что для работы с такой структурой понадобится конечное количество обходов, постоянное для каждой структуры. Говоря проще, для разных значений, переданных в функцию **fibonacci**, потребуется разное количество ее вызовов. Менять код под каждое передаваемое значение невозможно. Избавиться от этого недостатка помогает рекурсия.

# Практикум. Угадай число

Напишем первую игру — «Угадай число». Браузер будет загадывать случайное четырехзначное число, а мы будем отгадывать.

Попытки отгадать число будут идти через диалоговое окно — **prompt**. Браузер будет сообщать в ответ, больше или меньше загаданного наш ответ.

Алгоритм будет таким:

1. Браузер генерирует число и приглашает пользователя к игре.
2. Выводится окно запроса предположения.
3. Браузер проверяет число и возвращает результат.
4. Повторяем до тех пор, пока число не будет угадано.
5. Как только число угадано, браузер сбрасывает число попыток и генерирует новое число.

Пока не будем ничего выводить на страницу. И пока наш алгоритм далек от совершенства. Как только изучим новые возможности языка — сразу улучшим его.

# Практическое задание

1. Почему код дает именно такие результаты?

|  |
| --- |
| var a = 1, b = 1, c, d; c = ++a; alert(c); *// 2* d = b++; alert(d); *// 1* c = (2+ ++a); alert(c); *// 5* d = (2+ b++); alert(d); *// 4* alert(a); *// 3* alert(b); *// 3* |

1. Чему будет равен **x**?

|  |
| --- |
| var a = 2; var x = 1 + (a \*= 2); |

1. Объявить две целочисленные переменные — **a** и **b** и задать им произвольные начальные значения. Затем написать скрипт, который работает по следующему принципу:
   * если **a** и **b** положительные, вывести их разность;
   * если **а** и **b** отрицательные, вывести их произведение;
   * если **а** и **b** разных знаков, вывести их сумму;

Ноль можно считать положительным числом.

1. Присвоить переменной **а** значение в промежутке [0..15]. С помощью оператора **switch** организовать вывод чисел от **a** до 15.
2. Реализовать четыре основные арифметические операции в виде функций с двумя параметрами. Обязательно использовать оператор **return**.
3. Реализовать функцию с тремя параметрами: **function mathOperation(arg1, arg2, operation)**, где **arg1**, **arg2** — значения аргументов, operation — строка с названием операции. В зависимости от переданного значения выполнить одну из арифметических операций (использовать функции из пункта 5) и вернуть полученное значение (применить **switch**).
4. \* Сравнить **null** и **0**. Объяснить результат.
5. \* С помощью рекурсии организовать функцию возведения числа в степень. Формат: **function power(val, pow), где val** — заданное число, **pow** –— степень.

# Дополнительные материалы

1. [Рекурсия. Тренировочные задачи](https://habrahabr.ru/post/275813/).

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Дэвид Флэнаган. JavaScript. Подробное руководство.
2. Эрик Фримен, Элизабет Робсон. Изучаем программирование на JavaScript.