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)

[Домашнее задание](#_44sinio)

[Дополнительные материалы](#_ywyi7cn1thsm)

[Используемая литература](#_o592u646fkiw)

# Введение

Итак, 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, но работающую программу, выполняющую некие полезные действия. Но разумеется функционал языка гораздо шире.

Критерий истины есть практика. Поэтому новые знания, начиная с этого занятия, мы будем усваивать через реализацию игр.

# Операторы в JavaScript

Как и у любого языка программирования, в JavaScript также есть операторы. Сам по себе оператор – это наименьшая автономная часть языка программирования, т.е. команда. У операторов есть операнды. Операнд (или аргумент оператора) – это сущность, к которой применяется оператор. К примеру, при сложении двух чисел (3 + 2) работает оператор сложения с двумя операндами.

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

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

Бинарный же оператор применяется к двум операндам:

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

У некоторых операторов есть свои особые названия.

* Инкремент – 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 того, в каком порядке идут операнды.

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

Если же мы выполняем другие арифметические операторы, то такого приведения типов не будет – все операнды будут приводиться к числу.

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

Разумеется, Вы будете работать со сложными выражениями, содержащими более одного оператора. Тогда возникает необходимость расстановки приоритета операций, т.е. порядка их выполнения.

Если с арифметическими операторами всё просто – работает классическая логика (например, сначала умножение, потом сложение), то с программными операторами языка JavaScript всё несколько сложнее. Их приоритеты упорядочены в таблице, в которой операторы перечислены в порядке убывания приоритета.

|  |  |
| --- | --- |
| **Оператор** | **Описание** |
| . [ ] ( ) | Доступ к полям, индексация массивов, вызовы функций и группировка выражений. |
| ++ -- - ~ ! delete new typeof void | Унарные операторы, тип возвращаемых данных, создание объектов, неопределённые 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 происходит, казалось бы, очевидный процесс. При выполнении выражения

|  |
| --- |
| var a = 5 \* 3 – 7; |

сначала рассчитывается арифметическая часть выражения, а только потом произойдёт присвоение, т.к. оно находится ниже, чем сложение и умножение.

Также обратим внимание на унарные операторы инкрементрирования/декрементирования. В JavaScript есть префиксная и постфиксная форма их записи. По сути, обе формы увеличивают значение операнда на 1. Но давайте посмотрим, как они это делают.

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

Таким образом, в постфиксной форме сначала происходит возвращение значения, а потом выполняется инкрементирования. В префиксной форме инкрементирование производится сразу, а возврат происходит уже с обновлённым значением.

Разумеется, также в JS есть и операторы сравнения, которые возвращают логическое значение по выполнению оператора.

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

При побуквенном сравнении в случае, если сравниваются строки из нескольких букв, то сравнение осуществляется 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 ); |

Если Вы хотите, чтобы производилось строгое сравнение на равенство, то 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 равны чему бы то ни было ещё. Это жёсткое правило прописано в спецификации языка. При явном (т.е. при вызванном пользователем) преобразовании в число null принимает значение 0, а undefined - NaN.

# Принципы ветвления, визуализация, блок-схемы

В программном коде, как и в жизни, множество решений зависят от внешних факторов. И зависимость эта выражается в вербальном виде «Если случится событие А, то я выполню действие Б». Именно по такому принципу начинает строиться ветвление во всех языках программирования.

Как в русском языке для ветвления используется слово «если», в программировании применяются специальные операторы, обеспечивающие выполнение определённой команды или набора команд только при условии истинности логического выражения или группы выражений. Ветвление — одна из трёх (наряду с последовательным выполнением команд и циклом) базовых конструкций структурного программирования.

*Для справки*: в дискретной математике, которая является одной из фундаментальных наук, лежащих в основах программирования, условие ветвления есть предикат. Почитать об этом можно в дополнительной литературе.

Прежде чем приступать к написанию ветвлений на языке JavaScript, стоит поговорить о случаях, когда на ветвление влияет уйма факторов. В таком случае стоит визуализировать для себя логику программы или её части, чтобы не запутаться при реализации. Для решения задачи визуализации применяются так называемые блок-схемы.

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

**Процесс** (функцию обработки данных любого вида)

**Данные**



**Предопределенный процесс** (Символ отображает предопределенный процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте)

**Решение** (Это как раз то, о чём мы говорили в самом начале – ситуация, имеющая одну точку входа и ряд альтернативных выходов, один и только один из которых может быть использован после вычисления условий, определённых внутри этого символа.)





**Терминатор** (начало или конец программы)



Для наших целей на текущий момент перечисленных блоков вполне достаточно, а более подробный материал по блок-схемам можно найти, перейдя по прилагающейся к занятию ссылке.

Итак, для изучения ветвлений 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), т.е. такой вопрос, на который ответить можно только двумя способами: либо да, либо нет. Если выражение возвращает значение, отличное от типа boolean, то возвращаемое значение будет автоматически приведено к типу boolean: 0, null undefined, “” и NaN будут транслированы в false, остальные значения - в true. Действие выполняется тогда, когда условие истинно (true). Обычно условием является операция сравнения, либо несколько таких операций, объединённых логическими связками (И, ИЛИ). В результате проверки какого-либо условия может выполняться сразу несколько операторов:

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

Но что, если одного условия недостаточно? Рассмотрим пример ветвления, когда в случае истины мы выполним одно действие, а иначе – другое.





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

Давайте попробуем реализовать простой пример:

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

Обратите внимание на то, что если по условию нужно выполнять всего один оператор, то можно не ставить фигурные скобки.

Но не всегда можно уложить логику ветвления в две ветки. Но JS позволяет разделять нашу 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: нам даны два произвольных числа. Необходимо вывести на экран, их соотношение друг с другом. По сути, у нас будет 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 равен y"); |

## Оператор switch

Теперь представим ситуацию, в которой нам нужно разделить программу не на 2 или 3 варианта, а на большее количество. Конечно, мы можем много раз использовать конструкцию else if, но это может привести к серьёзному ухудшению читаемости кода. Поэтому существует специальный оператор выбора из нескольких вариантов – switch. Он имеет следующий синтаксис:

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

Оператор switch смотрит на значение переменной (вместо неё также может стоять выражение, возвращающее значение) и сравнивает его с предложенными вариантами. В случае совпадения выполняется соответствующий блок кода. Если же после прохода по всем вариантам совпадения так и не обнаружилось, то выполняются операторы из блока default. Это необязательный блок, и он 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. В случае, когда в конце блока case нет оператора break, интерпретатор продолжит 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:  echo '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). Звучит страшно, однако выглядит он достаточно просто:

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

Например, мы хотим сохранить максимальное из двух произвольных чисел в какую-то переменную. В этом случае, вместо громоздких строк ветвления, можно написать:

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

Тернарный оператор – красивая возможность, делающая код лаконичнее. Но, как и любым инструментом, не стоит злоупотреблять данной возможностью, наоборот усложняя код.

По своей сути тернарный оператор отличается от оператора if. Во-первых, недопустимо множественное использование тернарного оператора, как в случае if - else if. Это засоряет код. Во-вторых, тернарный оператор нужен для встраивания небольших условных веток прямо в выражение, т.е. он не заменяет собой стандартный if-else. В случае, если Вам необходимо описать условия непосредственно в выражении, то следует использовать тернарный оператор. Но если Вы хотите создать более сложное условие с телом, состоящим из более, чем одной инструкции, то Вы используете 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.

Таким образом

|  |
| --- |
| 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

Представим себе, что, используя код одного из примеров, мы хотим построить с пользователем диалог. Код в любой программе работает последовательно, строка за строкой. Таким образом условие уже отработано, вернуться к нему невозможно. Как решить эту задачу? Мы можем скопировать весь блок операций ещё несколько раз. Но что, если количество раз неизвестно заранее? Да и copy-paste – это уж совсем плохое решение. Тогда на помощь приходят функции.

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

Если проводить аналогию с реальной жизнью, то функция – это некий навык скрипта, который он знает и умеет делать. Ведь вы не учитесь ходить каждый раз, когда перемещаетесь между точками? Вы просто выполняете функцию «Ходить». Также и скрипт может иметь описанную функцию go, которая может вызываться в любой момент времени.

Функция в JS объявляется с помощью ключевого слова function. За ним следует название функции, 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 указания параметров в фигурных скобках следует тело функции. После объявления функции, мы можем её вызвать и посмотреть, как она работает. Описание функции может находиться и до, и после её вызова.

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

Давайте 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 неё нужно передавать такое количество параметров, которое заявили при её создании. Их может быть 0 и более. Если параметры не переданы, то 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, по сути, результатом её работы может являться только вывод какого-то текста на экран (см. предыдущую функцию). Однако, в большинстве случаев, мы хотим использовать результат работы функции в остальной программе. Тогда необходимо использовать оператор return. Например, напишем функцию, возвращающую среднее арифметическое двух чисел.

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

Таким образом, мы не только учим наш скрипт определённым навыкам, но и можем хранить результат выполнения каждой функции для дальнейшего использования.

## Области видимости

При работе с функциями в JS нужно также помнить о т.н. областях видимости. Они бывают глобальные и локальные. Глобальными называют переменные и функции, которые не находятся внутри какой-то функции.

В JS все глобальные переменные и функции являются свойствами специального объекта, который называется «глобальный объект» (global object). В браузере этот объект явно доступен под именем window. Объект window одновременно является глобальным объектом и содержит ряд свойств и методов для работы с окном браузера, но нас здесь интересует только его роль как глобального объекта.

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

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

## Рекурсия

Одним из наиболее интересных моментов в вопросе использования функций является рекурсия. Рекурсия – это вызов функцией самой себя. Когда это может быть полезно? Не стоит сейчас углубляться в решение задач обхода деревьев, но гораздо проще привести пример с вычислением последовательности n чисел Фибоначчи (каждое последующее число равно сумме двух предыдущих чисел). Каждый раз мы не знаем, сколько чисел Фибоначчи запросит пользователь, но, используя рекурсию, мы 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, которые имеют нефиксированное количество уровней вложенности, но на каждом уровне имеют жёсткую схему. Таким образом, Вы не можете сказать, что для работы с такой структурой Вам понадобится конечное количество обходов, постоянное для каждой структуры. Говоря проще, для разных значений, переданных в ту же функцию fibonacci, потребуется разное количество вызовов этой функции. Разумеется, менять код под каждое передаваемое значение невозможно. И избавиться от этого недостатка помогает рекурсия.

# Практикум. Угадай число

Теперь попробуем написать нашу первую игру. Начнём мы с простого и реализуем игру «Угадай число». Браузер будет загадывать случайное четырёхзначное число, а мы будем пытаться его отгадать.

Попытки отгадать число будут идти через диалоговое окно – prompt. Браузер будет сообщать в ответ, больше или меньше загаданного наше предположение.

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 мы не будем ничего выводить на саму страницу. И пока наш алгоритм будет далёк от совершенства. Но как только мы изучим новые возможности языка, то сразу улучшим его.

# Домашнее задание

1. Дан код:

|  |
| --- |
| 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 |

Почему код даёт именно такие результаты?

1. Чему будет равен x в примере ниже?

|  |
| --- |
| 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 основные 4 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 операции выполнить одну из арифметических операций (использовать функции из пункта 3) и вернуть полученное значение (использовать switch).
4. \* Сравнить null и 0. Попробуйте 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. <https://habrahabr.ru/post/275813/> - ещё о рекурсии

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

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