JavaScript

Основные особенности JavaScript, на уровне базовых конструкций, типов, синтаксиса.

Структура кода.

Операторы разделяются точкой с запятой:

```
alert('Привет'); alert('Мир');
```

Как правило, перевод строки тоже подразумевает точку с запятой. Так тоже будет работать:

```
alert('Привет')
alert('Мир')
```

…Однако, иногда JavaScript не вставляет точку с запятой. Например:

```
var a = 2
+3

alert(a); // 5
```

Переменные и типы.

- Объявляются директивой var. Могут хранить любое значение:

- ```
  var x = 5;
  x = "Петя";
  ```

- Есть 5 «примитивных» типов и объекты:

- ```
  x = 1;            // число
  x = "Тест";       // строка, кавычки могут быть одинарные или двойные
  x = true;         // булево значение true/false
  x = null;         // спец. значение (само себе тип)
  x = undefined;    // спец. значение (само себе тип)
  ```

  Также есть специальные числовые значения Infinity (бесконечность) и NaN.

  Значение NaN обозначает ошибку и является результатом числовой операции, если она некорректна.

- **Значение null не является «ссылкой на нулевой адрес/объект» или чем-то подобным. Это просто специальное значение.**

  Оно присваивается, если мы хотим указать, что значение переменной неизвестно.

  Например:

var age = null; // возраст неизвестен

- **Значение undefined означает «переменная не присвоена».**

  Например:

  var x;
  alert(x); // undefined

  Можно присвоить его и явным образом: x = undefined, но так делать не рекомендуется.

- В имени переменной могут быть использованы любые буквы или цифры, но цифра не может быть первой. Символы доллар $ и подчёркивание _ допускаются наравне с буквами.

Основные операторы

Для работы с переменными, со значениями, JavaScript поддерживает все стандартные операторы, большинство которых есть и в других языках программирования.

Несколько операторов мы знаем со школы – это обычные сложение +, умножение *, вычитание и так далее.

У операторов есть своя терминология, которая используется во всех языках программирования.

- *Операнд* – то, к чему применяется оператор. Например: 5 * 2 – оператор умножения с левым и правым операндами. Другое название: «аргумент оператора».

- *Унарным* называется оператор, который применяется к одному выражению. Например, оператор унарный минус "-" меняет знак числа на противоположный:

- *Бинарным* называется оператор, который применяется к двум операндам. Тот же минус существует и в бинарной форме:

Обычно при помощи плюса '+' складывают числа.

Но если бинарный оператор '+' применить к строкам, то он их объединяет в одну:

var a = "моя" + "строка";

alert(a); // моястрока

Иначе говорят, что «плюс производит конкатенацию (сложение) строк».

Если хотя бы один аргумент является строкой, то второй будет также преобразован к строке!

Причем не важно, справа или слева находится операнд-строка, в любом случае нестроковый аргумент будет преобразован. Например:

alert('1' + 2); // "12"

alert(2 + '1'); // "21"

Это приведение к строке – особенность исключительно бинарного оператора "+".

Остальные арифметические операторы работают только с числами и всегда приводят аргументы к числу.

Например:

alert(2 - '1'); // 1

alert(6 / '2'); // 3

Унарный, то есть применённый к одному значению, плюс ничего не делает с числами:

alert(+1); // 1

alert(+(1 - 2)); // -1

Как видно, плюс ничего не изменил в выражениях. Результат – такой же, как и без него.

Тем не менее, он широко применяется, так как его «побочный эффект» – преобразование значения в число.

Например, когда мы получаем значения из HTML-полей или от пользователя, то они обычно в форме строк.

А что, если их нужно, к примеру, сложить? Бинарный плюс сложит их как строки:

var apples = "2";

var oranges = "3";

alert(apples + oranges); // "23", так как бинарный плюс складывает строки

Поэтому используем унарный плюс, чтобы преобразовать к числу:

var apples = "2";

var oranges = "3";

alert(+apples + +oranges); // 5, число, оба операнда предварительно преобразованы в числа

С точки зрения математики такое изобилие плюсов может показаться странным. С точки зрения программирования – никаких разночтений: сначала выполнятся унарные плюсы, приведут строки к числам, а затем – бинарный '+' их сложит.

Почему унарные плюсы выполнились до бинарного сложения? Как мы сейчас увидим, дело в их приоритете.

В том случае, если в выражении есть несколько операторов – порядок их выполнения определяется *приоритетом*.

Из школы мы знаем, что умножение в выражении 2 * 2 + 1 выполнится раньше сложения, т.к. его *приоритет* выше, а скобки явно задают порядок выполнения. Но в JavaScript – гораздо больше операторов, поэтому существует целая таблица приоритетов.

Оператор присваивания =.

У него – один из самых низких приоритетов: 3.

Именно поэтому, когда переменную чему-либо присваивают, например, x = 2 * 2 + 1 сначала выполнится арифметика, а уже затем – произойдёт присваивание =.

```
var x = 2 * 2 + 1;

alert( x ); // 5
```

Возможно присваивание по цепочке:

```
var a, b, c;

a = b = c = 2 + 2;

alert( a ); // 4

alert( b ); // 4

alert( c ); // 4
```

Такое присваивание работает справа-налево, то есть сначала вычислятся самое правое выражение 2+2, присвоится в c, затем выполнится b = c и, наконец, a = b.

Оператор "=" возвращает значение

Все операторы возвращают значение. Вызов x = выражение не является исключением.

Он записывает выражение в x, а затем возвращает его. Благодаря этому присваивание можно использовать как часть более сложного выражения:

```
var a = 1;

var b = 2;
```

```
var c = 3 - (a = b + 1);

alert( a ); // 3

alert( c ); // 0
```

В примере выше результатом (a = b + 1) является значение, которое записывается в a (т.е. 3). Оно используется для вычисления c.

Забавное применение присваивания, не так ли?

Знать, как это работает – стоит обязательно, а вот писать самому – только если вы уверены, что это сделает код более читаемым и понятным.

Оператор взятия остатка % интересен тем, что, несмотря на обозначение, никакого отношения к процентам не имеет.

Его результат a % b – это остаток от деления a на b.

Например:

```
alert( 5 % 2 ); // 1, остаток от деления 5 на 2

alert( 8 % 3 ); // 2, остаток от деления 8 на 3

alert( 6 % 3 ); // 0, остаток от деления 6 на 3
```

Одной из наиболее частых операций в JavaScript, как и во многих других языках программирования, является увеличение или уменьшение переменной на единицу.

Для этого существуют даже специальные операторы:

- **Инкремент** ++ увеличивает на 1:

```
var i = 2;
i++;      // более короткая запись для i = i + 1.
alert(i); // 3
```

- **Декремент** -- уменьшает на 1:

```
var i = 2;
i--;      // более короткая запись для i = i - 1.
alert(i); // 1
```

Важно:

Инкремент/декремент можно применить только к переменной. Код 5++ даст ошибку.

Вызывать эти операторы можно не только после, но и перед переменной: i++ (называется «постфиксная форма») или ++i («префиксная форма»).

Обе эти формы записи делают одно и то же: увеличивают на 1.

Тем не менее, между ними существует разница. Она видна только в том случае, когда мы хотим не только увеличить/уменьшить переменную, но и использовать результат в том же выражении.

Например:

```
var i = 1;

var a = ++i; // (*)

alert(a); // 2
```

В строке (*) вызов ++i увеличит переменную, а *затем* вернёт ее значение в a. Так что в a попадёт значение i *после* увеличения.

Постфиксная форма i++ отличается от префиксной ++i тем, что возвращает старое значение, бывшее до увеличения.

В примере ниже в a попадёт старое значение i, равное 1:

```
var i = 1;

var a = i++; // (*)

alert(a); // 1
```

- Если результат оператора не используется, а нужно только увеличить/уменьшить переменную – без разницы, какую форму использовать:

- var i = 0;
- i++;
- ++i;
  alert(i); // 2

- Если хочется тут же использовать результат, то нужна префиксная форма:

- var i = 0;
  alert(++i); // 1

- Если нужно увеличить, но нужно значение переменной *до увеличения* – постфиксная форма:

- var i = 0;
  alert(i++); // 0

Инкремент/декремент можно использовать в любых выражениях

При этом он имеет более высокий приоритет и выполняется раньше, чем арифметические операции:

```
var i = 1;

alert( 2 * ++i ); // 4
```

```
var i = 1;

alert( 2 * i++ ); // 2, выполнился раньше но значение вернул старое
```

При этом, нужно с осторожностью использовать такую запись, потому что в более длинной строке при быстром «вертикальном» чтении кода легко пропустить такой i++, и будет неочевидно, что переменная увеличивается.

Три строки, по одному действию в каждой – длиннее, зато нагляднее:

```
var i = 1;

alert( 2 * i );

i++;
```

Часто нужно применить оператор к переменной и сохранить результат в ней же, например:

```
var n = 2;

n = n + 5;

n = n * 2;
```

Эту запись можно укоротить при помощи совмещённых операторов, вот так:

```
var 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
```

Так можно сделать для операторов +,-,*,/,% и бинарных <<,>>,>>>,&,|,^.

Вызов с присваиванием имеет в точности такой же приоритет, как обычное присваивание, то есть выполнится после большинства других операций:

```
var n = 2;

n *= 3 + 5;

alert( n ); // 16  (n = 2 * 8)
```

Один из самых необычных операторов – запятая ','.

Его можно вызвать явным образом, например:

```
var a = (5, 6);

alert( a );
```

Запятая позволяет перечислять выражения, разделяя их запятой ','. Каждое из них – вычисляется и отбрасывается, за исключением последнего, которое возвращается.

Запятая – единственный оператор, приоритет которого ниже присваивания. В выражении a = (5,6) для явного задания приоритета использованы скобки, иначе оператор '=' выполнился бы до запятой ',', получилось бы (a=5), 6.

Зачем же нужен такой странный оператор, который отбрасывает значения всех перечисленных выражений, кроме последнего?

Обычно он используется в составе более сложных конструкций, чтобы сделать несколько действий в одной строке. Например:

```
// три операции в одной строке

for (a = 1, b = 3, c = a*b; a < 10; a++) {

 ...

}
```

- **Сравнение === проверяет точное равенство, включая одинаковый тип.** Это самый очевидный и надёжный способ сравнения.

- **Остальные сравнения == < <= > >= осуществляют числовое приведение типа:**

- alert(0 == false); // true
  alert(true > 0); // true

  Исключение – сравнение двух строк, которое осуществляется лексикографически (см. далее).

  Также: значения null и undefined при == равны друг другу и не равны ничему ещё. А при операторах больше/меньше происходит приведение null к 0, а undefined к NaN.

  Такое поведение может привести к неочевидным результатам, поэтому лучше всего использовать для сравнения с null/undefined оператор ===. Оператор == тоже можно, если не хотите отличать null от undefined.

  Например, забавное следствие этих правил для null:

alert(null > 0); // false, т.к. null преобразовано к 0
alert(null >= 0); // true, т.к. null преобразовано к 0
alert(null == 0); // false, в стандарте явно указано, что null равен лишь undefined

С точки зрения здравого смысла такое невозможно. Значение null не равно нулю и не больше, но при этом null >= 0 возвращает true!

- **Сравнение строк – лексикографическое, символы сравниваются по своим unicode-кодам.**

  Поэтому получается, что строчные буквы всегда больше, чем прописные:

  alert('a' > 'Я'); // true

В JavaScript есть логические операторы: И (обозначается &&), ИЛИ (обозначается ||) и НЕ (обозначается !). Они интерпретируют любое значение как логическое.

Условные операторы: if, '?'

Иногда, в зависимости от условия, нужно выполнить различные действия. Для этого используется оператор if.

Синтаксис оператора if:

if (условие)

  {код, который работает, если условие выполнено}

либо

if (условие)

  {код, который выполняется, если условие выполнено}

else

  {код, который выполняется, если условие не выполнено}

Например:

var year = prompt('В каком году появилась спецификация ECMA-262 5.1?', '');

if (year != 2011) alert('А вот и неправильно!');

Оператор if (...) вычисляет и преобразует выражение в скобках к логическому типу.

В логическом контексте:

- Число 0, пустая строка "", null и undefined, а также NaN являются false,

- Остальные значения – true.

Например, такое условие никогда не выполнится:

```
if (0) { // 0 преобразуется к false
  ...
}
```

…А такое – выполнится всегда:

```
if (1) { // 1 преобразуется к true
  ...
}
```

Оператор вопросительный знак «?»

Иногда нужно в зависимости от условия присвоить переменную. Например:

```
var access;
var age = prompt('Сколько вам лет?', '');
if (age > 14) {
  access = true;
} else {
  access = false;
}
alert(access);
```

Оператор вопросительный знак '?' позволяет делать это короче и проще.

Он состоит из трех частей:

условие ? значение1 : значение2

Проверяется условие, затем если оно верно – возвращается значение1, если неверно – значение2, например:

```
access = (age > 14) ? true : false;
```

Оператор '?' выполняется позже большинства других, в частности – позже сравнений, поэтому скобки можно не ставить:

```
access = age > 14 ? true : false;
```

…Но когда скобки есть – код лучше читается. Так что рекомендуется их писать.

«Тернарный оператор»

Вопросительный знак – единственный оператор, у которого есть аж три аргумента, в то время как у обычных операторов их один-два. Поэтому его называют *«тернарный оператор»*.

Смысл оператора '?' – вернуть то или иное значение, в зависимости от условия. Пожалуйста, используйте его по назначению, а для выполнения разных веток кода есть if.

Конструкция switch заменяет собой сразу несколько if.

Она представляет собой более наглядный способ сравнить выражение сразу с несколькими вариантами.

Выглядит она так:

```
switch(x) {

  case 'value1':  // if (x === 'value1')

    ...

    [break]

  case 'value2':  // if (x === 'value2')

    ...

    [break]

  default:

    ...

    [break]

}
```

- Переменная x проверяется на строгое равенство первому значению value1, затем второму value2 и так далее.
- Если соответствие установлено – switch начинает выполняться от соответствующей директивы case и далее, *до ближайшего break* (или до конца switch).
- Если ни один case не совпал – выполняется (если есть) вариант default.

При этом case называют *вариантами switch*.

Пример использования switch (сработавший код выделен):

```
var a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Маловато' );
    break;
  case 4:
    alert( 'В точку!' );
    break;
  case 5:
    alert( 'Перебор' );
    break;
  default:
    alert( 'Я таких значений не знаю' );
}
```

Здесь оператор switch последовательно сравнит a со всеми вариантами из case.

Сначала 3, затем – так как нет совпадения – 4. Совпадение найдено, будет выполнен этот вариант, со строки alert('В точку!') и далее, до ближайшего break, который прервёт выполнение.

Если break нет, то выполнение пойдёт ниже по следующим case, при этом остальные проверки игнорируются.

Пример без break:

```
var a = 2 + 2;

switch (a) {
  case 3:
    alert( 'Маловато' );
  case 4:
    alert( 'В точку!' );
```

```
  case 5:

    alert( 'Перебор' );

  default:

    alert( 'Я таких значений не знаю' );

}
```

В примере выше последовательно выполнятся три alert:

```
alert( 'В точку!' );

alert( 'Перебор' );

alert( 'Я таких значений не знаю' );
```

В case могут быть любые выражения, в том числе включающие в себя переменные и функции.

Циклы

- Поддерживаются три вида циклов:

```
// 1
while (условие) {
  ...
}

// 2
do {
  ...
} while (условие);

// 3
for (var i = 0; i < 10; i++) {
  ...
}
```

- Переменную можно объявлять прямо в цикле, но видна она будет и за его пределами.

- Поддерживаются директивы break/continue для выхода из цикла/перехода на следующую итерацию.

Функции

Синтаксис функций в JavaScript:

```
// function имя(список параметров) {
тело } function sum(a, b) {
  var result = a + b;

  return result;
}

// использование:
alert( sum(1, 2) ); //
3
```

- sum – имя функции, ограничения на имя функции – те же, что и на имя переменной.
- Переменные, объявленные через var внутри функции, видны везде внутри этой функции, блоки if, for и т.п. на видимость не влияют.
- Параметры копируются в локальные переменные a, b.
- Функция без return считается возвращающей undefined. Вызов return без значения также возвращает undefined:
- function f() { }
  alert(f()); // undefined

Задание

а) Если уравнение $ax^2 + bx + c = 0\ (a \neq 0)$ имеет вещественные корни, то то логической переменной t присвоить значение значение true, а переменным x1 и x2 – сами корни, иначе же переменной t присвоить значение false, а значения переменных x1 и x2 не менять.

Объект Math предназначен для хранения некоторых математических констант и выполнения преобразований чисел с помощью типичных математических функций.
Math.PI – значение постоянной π.
Math.sqrt(x) - Возвращает квадратный корень из x.

б) Вычисление f= 10! Описать каждым из трех вариантов оператора цикла.

в) Есть **случайное число** N (от 1 до 100). Выведете на экран последовательность от 1 до N «елочкой», например для N=17:

```
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17
```

```
Возвращает случайное целое число между min (включительно) и
max (не включая max)
Math.floor(Math.random() * (max - min)) + min;
```

Примерный алгоритм.

N // случайное число

nStr=1 //номер строки

I=1 //число, которое печатаем

Пока i<=N

Начало цикла

Цикл (j от 1 до nStr) и i<=N

Начало цикла

Печатаем i +" "

Увеличиваем i на 1

Конец цикла

Переводим печать на другую строчку.

Увеличиваем nStr на 1

Конец цикла

г) Найти сумму и количество **элементов** последовательности, которые по модулю больше 0,001:

$$\mathbf{1}, \frac{\mathbf{2}}{\mathbf{3 \cdot 3}}, -\frac{\mathbf{4}}{\mathbf{5 \cdot 9}}, \frac{\mathbf{6}}{\mathbf{7 \cdot 27}}, -\frac{\mathbf{8}}{\mathbf{9 \cdot 81}}, \dots$$

ответ S = 1.157.

Делаем подобно.

Пример: Найти сумму всех элементов последовательности, которые по модулю больше 0,001.

$$1, -\frac{1}{2}, \frac{2}{4}, -\frac{3}{8}, \frac{4}{16}, -\frac{5}{32}, \dots$$

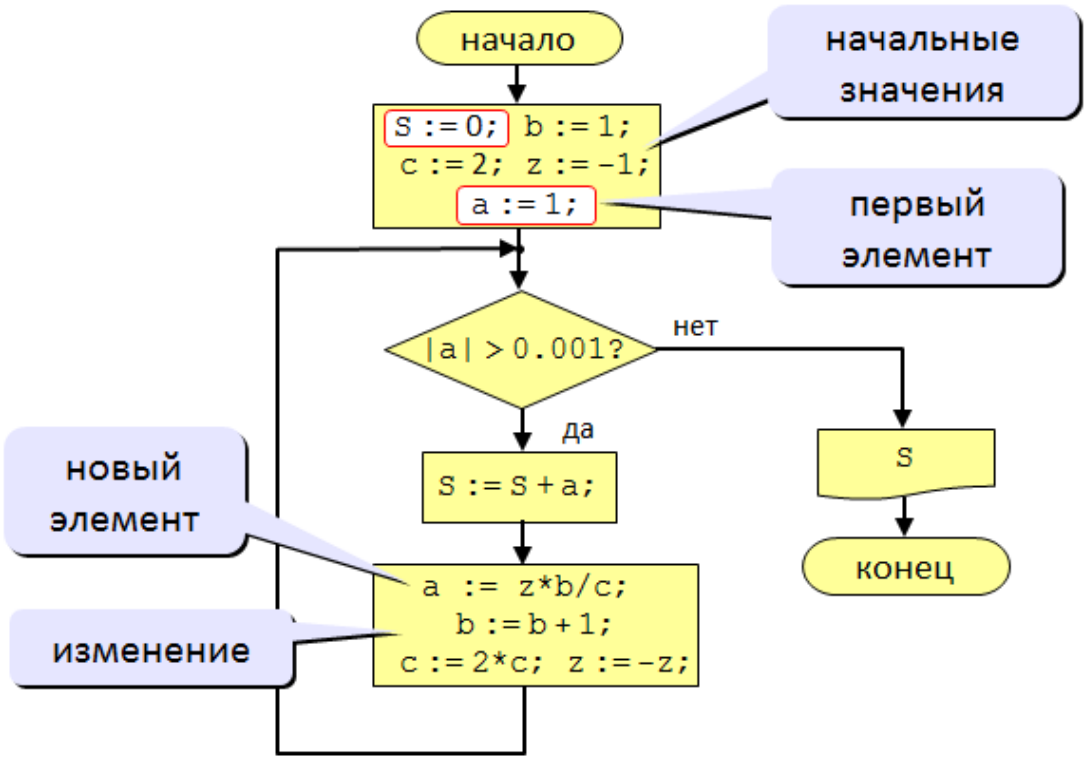
Элемент последовательности (начиная с №2):

$$a = z\,\frac{b}{c}$$

n	1	2	3	4	5	...
b	1	2	3	4	5	...
c	2	4	8	16	32	...
z	-1	1	-1	1	-1	...

```
b := b+1;
c := 2*c;
z := -z;
```

Алгоритм


начало
S := 0; b := 1;
c := 2; z := -1;
a := 1;
начальные значения
первый элемент
|a| > 0.001?
нет
да
S := S + a;
a := z*b/c;
b := b + 1;
c := 2*c; z := -z;
новый элемент
изменение
S
конец