# Что такое JavaScript?

JavaScript изначально создавался для того, чтобы сделать web-странички «живыми».

Программы на этом языке называются скриптами. Они подключаются напрямую к HTML и, как только загружается страничка — тут же выполняются.

Программы на JavaScript — обычный текст. Они не требуют какой-то специальной подготовки.

В этом плане JavaScript сильно отличается от другого языка, который называется Java.

## Почему JavaScript?

Когда создавался язык JavaScript, у него изначально было другое название: «LiveScript». Но тогда был очень популярен язык Java, и маркетологи решили, что схожее название сделает новый язык более популярным.

Планировалось, что JavaScript будет эдаким «младшим братом» Java. Однако, история распорядилась по-своему, JavaScript сильно вырос, и сейчас это совершенно независимый язык, со своей спецификацией, которая называется ECMAScript, и к Java не имеет никакого отношения.

У него много особенностей, которые усложняют освоение, но по ходу учебника мы с ними разберемся.

Чтобы читать и выполнять текст на JavaScript, нужна специальная программа — интерпретатор. Процесс выполнения скрипта называют «интерпретацией».

Во все основные браузеры встроен интерпретатор JavaScript, именно поэтому они могут выполнять скрипты на странице.

Но, разумеется, этим возможности JavaScript не ограничены. Это полноценный язык, программы на котором можно запускать и на сервере, и даже в стиральной машинке, если в ней установлен соответствующий интерпретатор.

## Что умеет JavaScript?

Современный JavaScript — это «безопасный» язык программирования общего назначения. Он не предоставляет низкоуровневых средств работы с памятью, процессором, так как изначально был ориентирован на браузеры, в которых это не требуется.

В браузере JavaScript умеет делать все, что относится к манипуляции со страницей, взаимодействию с посетителем и, в какой-то мере, с сервером:

• Создавать новые HTML-теги, удалять существующие, менять стили элементов, прятать, показывать элементы и т.п.

• Реагировать на действия посетителя, обрабатывать клики мыши, перемещение курсора, нажатие на клавиатуру и т.п.

• Посылать запросы на сервер и загружать данные без перезагрузки страницы(это технология называется "AJAX").

• Получать и устанавливать cookie, запрашивать данные, выводить сообщения…

• …и многое, многое другое!

## Что НЕ умеет JavaScript?

JavaScript — быстрый и мощный язык, но браузер накладывает на его исполнение некоторые ограничения.

Это сделано для безопасности пользователей, чтобы злоумышленник не мог с помощью JavaScript получить личные данные или как-то навредить компьютеру пользователя. В браузере Firefox существует способ «подписи» скриптов с целью обхода части ограничений, но он нестандартный и не кросс-браузерный.

Этих ограничений нет там, где JavaScript используется вне браузера, например на сервере.

Большинство возможностей JavaScript в браузере ограничено текущим окном и страницей.

• JavaScript не может читать/записывать произвольные файлы на жесткий диск, копировать их или вызывать программы. Он не имеет прямого доступа к операционной системе.

Современные браузеры могут работать с файлами, но эта возможность ограничена специально выделенной директорией — песочницей. Возможности по доступу к устройствам также прорабатываются в современных стандартах и, частично, доступны в некоторых браузерах.

• JavaScript, работающий в одной вкладке, почти не может общаться с другими вкладками и окнами. За исключением случая, когда он сам открыл это окно или несколько вкладок из одного источника (одинаковый домен, порт, протокол).

Есть способы это обойти, и они раскрыты в учебнике, но для этого требуется как минимум явное согласие обеих сторон. Просто так взять и залезть в произвольную вкладку с другого домена нельзя.

• Из JavaScript можно легко посылать запросы на сервер, с которого пришла страничка. Запрос на другой домен тоже возможен, но менее удобен, т.к. и здесь есть ограничения безопасности.

## В чем уникальность JavaScript?

Есть как минимум три замечательных особенности JavaScript:

1. Полная интеграция с HTML/CSS.

2. Простые вещи делаются просто.

3. Поддерживается всеми распространенными браузерами и включен по умолчанию.

Этих трех вещей одновременно нет больше ни в одной браузерной технологии. Поэтому JavaScript и является самым распространенным средством создания браузерных интерфейсов.

## Тег SCRIPT

Программы на языке JavaScript можно вставить в любое место HTML при помощи тега SCRIPT. Например:

<!DOCTYPE HTML>

<html>

<head>

<!-- Тег meta для указания кодировки -->

<meta charset="utf-8">

</head>

<body>

<p>Начало документа...</p>

<script>

alert('Привет, Мир!');

</script>

<p>...Конец документа</p>

</body>

</html>

Этот пример использует следующие элементы:

<script> ... </script>

Тег script содержит исполняемый код. Предыдущие стандарты HTML требовали обязательного указания атрибута type, но сейчас он уже не нужен. Достаточно просто<script>.

Браузер, когда видит <script>:

1. Начинает отображать страницу, показывает часть документа до script

2. Встретив тег script, переключается в JavaScript-режим и не показывает, а исполняет его содержимое.

3. Закончив выполнение, возвращается обратно в HTML-режим и отображает оставшуюся часть документа.

Попробуйте этот пример в действии, обратите внимание что пока браузер не выполнит скрипт - он не может отобразить часть страницы после него.

alert(...)

Отображает окно с сообщением и ждет, пока посетитель не нажмет «Ок»

## Внешние скрипты

Если JavaScript-кода много — его выносят в отдельный файл, который подключается в HTML:

<script src="/path/to/script.js"></script>

Здесь /path/to/script.js - это абсолютный путь к файлу, содержащему скрипт (из корня сайта).

Браузер сам скачает скрипт и выполнит.

Например:

<html>

<head>

<meta charset="utf-8">

<script src="/files/tutorial/browser/script/rabbits.js"></script>

</head>

<body>

<script>

count\_rabbits();

</script>

</body>

</html>

Содержимое файла /files/tutorial/browser/script/rabbits.js:

function count\_rabbits()

{

for(var i=1; i<=3; i++)

{

alert("Кролик номер "+i);

}

}

Можно указать и полный URL, например:

<script src="http://code.jquery.com/jquery.js"></script>

Вы также можете использовать путь относительно текущей страницы, например src="script.js"если скрипт находится в том же каталоге, что и страница.

Чтобы подключить несколько скриптов, используйте несколько тегов:

<script src="/js/script1.js"></script>

<script src="/js/script2.js"></script>

Как правило, в HTML пишут только самые простые скрипты, а сложные выносят в отдельный файл.

Благодаря этому один и тот же скрипт, например, меню или библиотека функций, может использоваться на разных страницах.

Браузер скачает его только первый раз и в дальнейшем, при правильной настройке сервера, будет брать из своего кэша.

Если указан атрибут src, то содержимое тега игнорируется.

В одном теге SCRIPT нельзя одновременно подключить внешний скрипт и указать код.

Вот так не cработает:

<script src="file.js">

alert(1); // если указан src, то внутренняя часть скрипта игнорируется

</script>

Нужно выбрать: либо SCRIPT идёт с src, либо содержит код. Тег выше следует разбить на два: один — с src, другой с кодом:

<script src="file.js"></script>

<script>

alert(1);

</script>

Например, можно вместо одного вызова alert сделать два:

alert('Привет'); alert('Мир');

Как правило, новая команда занимает отдельную строку — так код лучше читается:

alert('Привет');

alert('Мир');

Точку с запятой во многих случаях можно не ставить, если есть переход на новую строку. Так тоже будет работать:

alert('Привет')

alert('Мир')

В этом случае JavaScript интерпретирует переход на новую строчку как разделитель команд и автоматически вставляет «виртуальную» точку с запятой между ними.

Однако, внутренние правила по вставке точки с запятой не идеальны. В примере выше они сработали, но в некоторых ситуациях JavaScript «забывает» вставить точку с запятой там, где она нужна. Таких ситуаций не так много, но они все же есть, и ошибки, которые при этом появляются, достаточно сложно исправлять.

Поэтому рекомендуется точки с запятой ставить. Сейчас это, фактически, стандарт.

## Комментарии

Со временем программа становится большой и сложной. Появляется необходимость добавить комментарии, которые объясняют, что происходит и почему.

Комментарии могут находиться в любом месте программы и никак не влияют на ее выполнение. Интерпретатор JavaScript попросту игнорирует их.

Однострочные комментарии начинаются с двойного слэша //. Текст считается комментарием до конца строки:

// Команда ниже говорит "Привет"

alert('Привет');

alert('Мир'); // Второе сообщение выводим отдельно

Многострочные комментарии начинаются слешем-звездочкой "/\*" и заканчиваются звездочкой-слэшем "\*/", вот так:

/\* Пример с двумя сообщениями.

Это - многострочный комментарий.

\*/

alert('Привет');

alert('Мир');

Все содержимое комментария игнорируется. Если поместить код внутрь /\* ... \*/ или после // — он не выполнится.

/\* Закомментировали код

alert('Привет');

\*/

alert('Мир');

## Переменная

Переменная состоит из имени и выделенной области памяти, которая ему соответствует.

Для объявления или, другими словами, создания переменной используется ключевое слово var:

var message;

После объявления, можно записать в переменную данные:

var message;

message = 'Привет'; // сохраним в переменной строку

Эти данные будут сохранены в соответствующей области памяти и в дальнейшем доступны при обращении по имени:

var message;

message = 'Привет';

alert(message); // выведет содержимое переменной

Для краткости можно совместить объявление переменной и запись данных:

var message = 'Привет';

## Аналогия из жизни

Проще всего понять переменную, если представить ее как «коробку» для данных, с уникальным именем.

Например, переменная message - это коробка, в которой хранится значение "Привет"

В коробку можно положить любое значение, а позже - поменять его. Значение в переменной можно изменять сколько угодно раз

var message;

message = 'Привет';

message = 'Мир'; // заменили значение

alert(message);

При изменении значения старое содержимое переменной удаляется.

## Копирование значений

Переменные в JavaScript могут хранить не только строки, но и другие данные, например, числа.

Объявим две переменные, положим в одну - строку, а в другую - число.

Как вы можете видеть, переменной без разницы, что хранить:

var num = 100500;

var message = 'Привет';

Значение можно копировать из одной переменной в другую.

var num = 100500;

var message = 'Привет';

message = num;

Значение из num перезаписывает текущее в message.

После этого присваивания в обеих коробках num и message находится одно и то же значение 100500.

## Важность директивы var

В JavaScript вы можете создать переменную и без var, достаточно просто присвоить ей значение:

x = "value"; // переменная создана, если ее не было

Технически, это не вызовет ошибки, но делать так все-таки не стоит.

Всегда определяйте переменные через var. Это хороший тон в программировании и помогает избежать ошибок.

Откройте пример в IE в новом окне:

<html>

<head>

<meta http-equiv="X-UA-Compatible" content="IE=8">

</head>

<body>

<div id="test"></div>

<script>

test = 5;

alert(test);

</script>

</body>

</html>

Значение не выведется, будет ошибка. Если в IE включена отладка или открыта панель разработки - вы увидите ее.

Дело в том, что почти все современные браузеры создают переменные для элементов, у которых есть id.

Переменная test в них в любом случае существует, запустите, к примеру:

<div id="test"></div>

<script>

alert(test); // выведет элемент

</script>

..Но в IE<9 такую переменную изменять нельзя.

Всё будет хорошо, если объявить test, используя var:

Правильный код:

<html>

<body>

<div id="test"></div>

<script>

var test = 5;

alert(test);

</script>

</body>

</html>

Самое «забавное» — то, что, эта ошибка будет только в IE<9, и только если на странице присутствует элемент с совпадающим id.

Такие ошибки особенно весело исправлять и отлаживать.

Есть и еще ситуации, когда отсутствие var может привести к ошибкам. Надеюсь, вы убедились в необходимости всегда ставить var.

## Константы

Константа — это переменная, которая никогда не меняется. Как правило, их называют большими буквами, через подчёркивание. Например:

var COLOR\_BLUE = "#00F";

var COLOR\_RED = "#0F0";

var COLOR\_GREEN = "#F00";

var COLOR\_ORANGE = "#FF7F00";

alert(COLOR\_RED); // #0F0

Технически, константа является обычной переменной, то есть её можно изменить. Но мы договариваемся этого не делать.

Зачем нужны константы? Почему бы просто не использовать "#F00" или "#0F0"?

1. Во-первых, константа — это понятное имя, в отличие от строки "#FF7F00".

2. Во-вторых, опечатка в строке может быть не замечена, а в имени константы её упустить невозможно — будет ошибка при выполнении.

Константы используют вместо строк и цифр, чтобы сделать программу понятнее и избежать ошибок.

## Имена переменных

На имя переменной наложены два ограничения.

1. Имя может состоять из: букв, цифр, символов $ и \_

2. Первый символ не должен быть цифрой.

Например:

var myName;

var test123;

Что здесь особенно интересно - доллар '$' и знак подчеркивания '\_' являются такими же обычными символами, как буквы:

var $ = 5; // объявили переменную с именем '$'

var \_ = 15; // переменная с именем '\_'

alert($);

А такие переменные были бы неправильными:

var 1a; // начало не может быть цифрой

var my-name; // дефис '-' не является разрешенным символом

Регистр букв имеет значение

Переменные apple и AppLE - две разные переменные.

## Зарезервированные имена

Существует список зарезервированных слов, которые нельзя использовать при именовании переменных, так как они используются самим языком, например: var, class, return, implementsи др.

Некоторые слова, например, class, не используются в современном JavaScript, но они заняты на будущее. Некоторые браузеры позволяют их использовать, но это может привести к ошибкам.

Следующий пример будет работать во многих старых браузерах, которые допускают использование слова 'class' и не сработает в современных. Они выдадут синтаксическую ошибку, попробуйте, запустите:

var class = 5;

alert(class);

## Правильный выбор имени

Правильный выбор имени переменной - одна из самых важных и сложных вещей в программировании, которая отличает начинающего от гуру.

Дело в том, что большинство времени мы тратим не на изначальное написание кода, а на его развитие.

А что такое развитие? Это когда я вчера написал код, а сегодня (или спустя неделю) прихожу и хочу его поменять. Например, вывести сообщение не так, а эдак.. Обработать товары по-другому, добавить функционал.. А где у меня там сообщение хранится?..

Гораздо проще найти нужные данные, если они правильно помечены, т.е. переменная названаправильно.

• Правило 1. Никакого транслита. Только английский.

Неприемлемы:

var moiTovari;

var cena;

var ssilka;

Подойдут:

var myGoods;

var price;

var link;

Если вы вдруг не знаете английский - самое время выучить. Все равно ведь придется…

• Правило 2. Использовать короткие имена только для переменных «местного значения».

Называть переменные именами, не несущими смысловой нагрузки, например a, e, p, mg - можно только в том случае, если они используются в небольшом фрагменте кода и их применение очевидно.

Вообще же, название переменной должно быть понятным. Иногда для этого нужно использовать несколько слов.

• Правило 3. Переменные из нескольких слов пишутся вместеВотТак.

Например:

var borderLeftWidth;

Этот способ записи называется «верблюжьей нотацией» или, по-английски, «camelCase».

Существует альтернативный стандарт, когда несколько слов пишутся через знак подчеркивания'\_':

var border\_left\_width;

Преимущественно в JavaScript используется вариант borderLeftWidth, в частности во встроенных языковых и браузерных функциях. Поэтому целесообразно остановиться на нём.

Ещё одна причина выбрать «верблюжью нотацию» — запись в ней немного короче, чем c подчеркиванием, т.к. не нужно вставлять '\_'.

• Правило последнее, главное. Имя переменной должно максимально четко соответствовать хранимым в ней данным.

Придумывание таких имен - одновременно коротких и точных, приходит с опытом, но только если сознательно стремиться к этому.

Позвольте поделиться одним небольшим секретом, который позволит улучшить ваши названия переменных и сэкономит вам время.

Бывает так, что вы написали код, через некоторое время к нему возвращаетесь, и вам надо что-то поправить. Например, изменить какую-то рамку «border…»

… И вы помните, что переменная, в которой хранится нужное вам значение, называется примерно так: borderLeftWidth. Вы ищете ее в коде, не находите, разбираетесь, и обнаруживаете, что на самом деле переменная называлась вот так: leftBorderWidth. После чего вносите нужные правки.

В этом случае, самый лучший ход - это переименовать переменную на ту, которую вы искали изначально. То есть, у вас в коде leftBorderWidth, а вы ее переименовываете наborderLeftWidth.

Зачем? Дело в том, что в следующий раз, когда вы захотите что-то поправить, то вы будете искать по тому же самому имени. Соответственно, это сэкономит вам время.

Кроме того, поскольку именно это имя переменной пришло вам в голову - скорее всего, оно больше соответствует хранимым там данным, чем то, которое вы придумали изначально.

Смысл имени переменной - это «имя на коробке», по которому мы сможем максимально быстро находить нужные нам данные.

Не нужно бояться переименовывать переменные, если вы придумали имя получше. Современные редакторы позволяют делать это очень удобно. Это в конечном счете сэкономит вам время.

Храните в переменной то, что следует

Бывают ленивые программисты, которые, вместо того чтобы объявить новую переменную, используют существующую.

В результате получается, что такая переменная — как коробка, в которую кидают то одно, то другое, то третье, при этом не меняя название. Что в ней лежит сейчас? А кто его знает.. Нужно подойти, проверить.

Сэкономит такой программист время на объявлении переменной — потеряет в два раза больше на отладке кода.

«Лишняя» переменная — добро, а не зло.

## Типы данных

Число number:

var n = 123;

n = 12.345;

Единый тип число используется как для целых, так и для дробных чисел.

Существуют специальные числовые значения Infinity (бесконечность) и NaN (ошибка вычислений). Они также принадлежат типу «число».

Например, бесконечность Infinity получается при делении на ноль:

alert( 1 / 0 ); // Infinity

Ошибка вычислений NaN будет результатом некорректной математической операции, например:

alert( "нечисло" \* 2 ); // NaN, ошибка

Строка string:

var str = "Мама мыла раму";

str = 'Одинарные кавычки тоже подойдут';

В JavaScript одинарные и двойные кавычки равноправны. Можно использовать или те или другие.

Тип символ не существует, есть только строка

В некоторых языках программирования есть специальный тип данных для одного символа. Например, в языке С это char. В JavaScript есть только тип «строка» string. Что, надо сказать, вполне удобно..

Булевый (логический) тип boolean. У него всего два значения - true (истина) и false (ложь).

Как правило, такой тип используется для хранения значения типа да/нет, например:

var checked = true; // поле формы помечено галочкой

checked = false; // поле формы не содержит галочки

Мы поговорим более подробно, когда будем обсуждать логические вычисления и условные операторы.

null — специальное значение. Оно имеет смысл «ничего». Значение null не относится ни к одному из типов выше, а образует свой отдельный тип, состоящий из единственного значения null:

var age = null;

В JavaScript null не является «ссылкой на несуществующий объект» или «нулевым указателем», как в некоторых других языках. Это просто специальное значение, которое имеет смысл «ничего» или «значение неизвестно».

В частности, код выше говорит о том, что возраст age неизвестен.

undefined — специальное значение, которое, как и null, образует свой собственный тип. Оно имеет смысл «значение не присвоено».

Если переменная объявлена, но в неё ничего не записано, то ее значение как раз и есть undefined:

var u;

alert(u); // выведет "undefined"

Можно присвоить undefined и в явном виде, хотя это делается редко:

var x = 123;

x = undefined;

В явном виде undefined обычно не присваивают, так как это противоречит его смыслу. Для записи в переменную «пустого значения» используется null.

Объекты object.

Первые 5 типов называют «примитивными».

Особняком стоит шестой тип: «объекты». К нему относятся, например, даты, он используется для коллекций данных и для многого другого.

## Термины: «унарный», «бинарный», «операнд»

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

• 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:

var x = 1;

alert( -x ); // -1, унарный минус

alert( -(x+2) ); // -3, унарный минус применён к результату сложения x+2

alert( -(-3) ); // 3

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

var x = 1, y = 3;

alert( y - x ); // 2, бинарный минус

Работа унарного "+" и бинарного "+" в JavaScript существенно различается.

Это действительно разные операторы. Бинарный плюс складывает операнды, а унарный — ничего не делает в арифметическом плане, но зато приводит операнд к числовому типу. Далее мы увидим примеры.

## Арифметические операторы

Базовые арифметические операторы знакомы нам с детства: это плюс +, минус -, умножить \*, поделить /.

Например:

alert(2 + 2); // 4

Или чуть сложнее:

var i = 2;

i = (2 + i) \* 3 / i;

alert(i); // 6

Более редкий арифметический оператор % интересен тем, что 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

## Сложение строк, бинарный +

Если бинарный оператор + применить к строкам, то он их объединяет в одну:

var a = "моя" + "строка";

alert(a); // моястрока

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 бинарного оператора "+".

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

Например:

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

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

## Унарный плюс +

Унарный плюс как арифметический оператор ничего не делает:

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, у нас есть два числа, в форме строк, и нужно их сложить. Бинарный плюс сложит их как строки, поэтому используем унарный плюс, чтобы преобразовать к числу:

var a = "2";

var b = "3";

alert( a + b ); // 23, так как бинарный плюс складывает строки

alert( +a + b ); // 23, второй операнд - всё ещё строка

alert( +a + +b); // 5, оба операнда предварительно преобразованы в числа

## Присваивание

Оператор присваивания выглядит как знак равенства =:

var i = 1 + 2;

alert(i); // 3

Он вычисляет выражение, которое находится справа, и присваивает результат переменной. Это выражение может быть достаточно сложным и включать в себя любые другие переменные:

var a = 1;

var b = 2;

a = b + a + 3; // (\*)

alert(a); // 6

В строке (\*) сначала произойдет вычисление, использующее текущее значение a (т.е. 1), после чего результат перезапишет старое значение a.

Возможно присваивание по цепочке:

var a, b, c;

a = b = c = 2 + 2;

Такое присваивание работает справа-налево, то есть сначала вычислятся самое правое выражение2+2, присвоится в c, затем выполнится b = c и, наконец, a = b.

Оператор "=" возвращает значение

Все операторы возвращают значение. Вызов x = выражение 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.

## Приоритет

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.

Она содержит как уже пройденные операторы, так и те, которые мы еще не проходили. В ней каждому оператору задан числовой приоритет. Тот, у кого число меньше — выполнится раньше. Если приоритет одинаковый, то порядок выполнения — слева направо.

Отрывок из таблицы:

… … …

5 умножение \*

5 деление /

6 сложение +

6 вычитание -

17 присвоение =

… … …

Посмотрим на таблицу в действии.

В выражении x = 2 \* 2 + 1 приоритет умножения \* равен 5, он самый высокий, поэтому выполнится раньше всех. Затем произойдёт сложение +, у которого приоритет 6, и после них — присвоение =, с приоритетом 17.

## Инкремент/декремент: ++, --

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

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. То есть, 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 при чтении кода зачастую неочевидно, что переменая увеличивается. Три строки — длиннее, зато нагляднее:

var i = 1;

alert( 2 \* i );

i++;

## Логические значения

Как и другие операторы, сравнение возвращает значение. Это значение имеет специальный логический тип.

Существует всего два логических значения:

• true — имеет смысл «да», «верно», «истина».

• false — означает «нет», «неверно», «ложь».

Например:

alert( 2 > 1 ); // true, верно

alert( 2 == 1 ); // false, неверно

alert( 2 != 1 ); // true

Логические значения можно использовать и напрямую, присваивать переменным, работать с ними как с любыми другими:

var a = true; // присвоили явно

var b = 3 > 4; // false

alert( b ); // false

alert( a == b ); // (true == false) неверно, результат false

## Сравнение строк

Строки сравниваются побуквенно:

alert( 'Б' > 'А' ); // true

Буквы сравниваются в алфавитном порядке. Какая буква в алфавите позже — та и больше.

## Кодировка Unicode

Аналогом «алфавита» во внутреннем представлении строк служит кодировка, у каждого символа — свой номер (код). JavaScript использует кодировку Unicode. При этом сравниваются численные коды символов.

В кодировке Unicode обычно код у строчной буквы больше, чем у прописной, поэтому:

alert('а' > 'Я'); // true, строчные буквы больше больших

Для корректного сравнения символы должны быть в одинаковом регистре.

Сравнение осуществляется как в телефонной книжке или в словаре. Сначала сравниваются первые буквы, потом вторые, и так далее, пока одна не будет больше другой.

Иными словами, больше — та строка, которая в телефонной книге была бы на большей странице.

Например:

• Если первая буква одной строки больше — значит первая строка больше, независимо от остальных символов:

alert( 'Банан' > 'Аят' );

• Если одинаковы — сравнение идёт дальше. Здесь оно дойдёт до третьей буквы:

alert( 'Вася' > 'Ваня' ); // true, т.к. 'с' > 'н'

• При этом любая буква больше отсутствия буквы:

alert( 'Привет' > 'Прив' ); // true, так как 'е' больше чем "ничего".

Такое сравнение называется лексикографическим.

Обычно мы получаем значения от посетителя в виде строк. Например, prompt возвращает строку, которую ввел посетитель.

Числа, полученные таким образом, в виде строк сравнивать нельзя, результат будет неверен. Например:

alert( "2" > "14" ); // true

В примере выше 2 оказалось больше 14, потому что строки сравниваются посимвольно, а первый символ '2' больше '1'.

Правильно было бы преобразовать их к числу явным образом. Например, поставив перед ними +:

alert( +"2" > +"14" ); // false, правильно

## Сравнение разных типов

При сравнении значения преобразуются к числам. Исключение: когда оба значения — строки, тогда не преобразуются.

Например:

alert( '2' > 1 ); // true

alert( '01' == 1 ); //true

alert( false == 0 ); // true, false становится 0, а true 1.

## Строгое равенство

Обычное равенство не может отличить 0 от false:

alert(0 == false); // true, т.к. false преобразуется к 0

Что же делать, если все же нужно отличить 0 от false?

Для проверки равенства без преобразования типов используются операторы строгого равенства === (тройное равно) и !==.

Они сравнивают без приведения типов. Если тип разный, то такие значения всегда неравны (строго):

alert(0 === false); // false, т.к. типы различны

## Сравнение с null и undefined

Проблемы со специальными значениями возможны, когда к переменной применяется операция сравнения > < <= >=, а у неё может быть как численное значение, так и null/undefined.

Интуитивно, кажется что null/undefined эквивалентны нулю, но это не так! Они ведут себя по-другому.

1. Значения null и undefined равны == друг другу и не равны чему бы то ни было ещё.

Это жёсткое правило буквально прописано в спецификации языка.

2. При преобразовании в число null становится 0, а undefined становится NaN.

Посмотрим забавные следствия.

Некорректный результат сравнения null с 0

Сравним null с нулём:

alert(null > 0); // false

alert(null == 0); // false

Итак, мы получили, что null не больше и не равен нулю. А теперь…

alert(null >= 0); // true

Как такое возможно? Если нечто «больше или равно нуля», то резонно полагать, что оно либо больше, либо равно. Но здесь это не так.

Дело в том, что алгоритмы проверки равенства == и сравнения >= > < <= работают по-разному.

Сравнение честно приводит к числу, получается ноль. А при проверке равенства значения null и undefined обрабатываются особым образом: они равны друг другу, но не равны чему-то ещё.

В результате получается странная с точки зрения здравого смысла ситуация, которую мы видели в примере выше.

## Несравнимый undefined

Значение undefined вообще нельзя сравнивать:

alert(undefined > 0); // false (1)

alert(undefined < 0); // false (2)

alert(undefined == 0); // false (3)

• Сравнения (1) и (2) дают false потому, что undefined при преобразовании к числу даёт NaN. А значение NaN по стандарту устроено так, что любые сравнения с ним возвращают false.

• Проверка равенства (3) даёт false, потому что в стандарте явно прописано, что undefined равно лишь null и ничему другому.

Вывод: любые сравнения с undefined/null, кроме точного ===, следует делать с осторожностью. Желательно не использовать сравнения >= > < <=, во избежание ошибок в коде.

## alert

Синтаксис:

alert(сообщение)

alert выводит на экран окно с сообщением и приостанавливает выполнение скрипта, пока пользователь не нажмет «ОК».

alert("Привет");

Окно сообщения, которое выводится, является модальным окном. Слово «модальное» означает, что посетитель не может взаимодействовать со страницей, нажимать другие кнопки и т.п., пока не разберется с окном. В данном случае - пока не нажмет на «OK».

## prompt

Функция prompt принимает два аргумента:

result = prompt(title, default);

Она выводит модальное окно с заголовком title, полем для ввода текста, заполненным строкой по умолчанию default и кнопками OK/CANCEL.

Пользователь должен либо что-то ввести и нажать OK, либо отменить ввод кликом на CANCEL или нажатием ESC на клавиатуре.

Вызов prompt возвращает то, что ввел посетитель - строку или специальное значение null, если ввод отменен.

Как и в случае с alert, окно prompt модальное.

var years = prompt('Сколько вам лет?', 100);

alert('Вам ' + years + ' лет!')

Всегда указывайте default

Вообще, второй default может отсутствовать. Однако при этом IE вставит в диалог значение по умолчанию "undefined".

Запустите этот код в IE, чтобы понять о чем речь:

var test = prompt("Тест");

Поэтому рекомендуется всегда указывать второй аргумент:

var test = prompt("Тест", ''); // <-- так лучше

## confirm

Синтаксис:

result = confirm(question);

confirm выводит окно с вопросом question с двумя кнопками: OK и CANCEL.

Результатом будет true при нажатии OK и false - при CANCEL(Esc).

Например:

var isAdmin = confirm("Вы - администратор?");

alert(isAdmin);

## Особенности встроенных функций

Место, где выводится модальное окно с вопросом, и внешний вид окна выбирает браузер. Разработчик не может на это влиять.

С одной стороны — это недостаток, т.к. нельзя вывести окно в своем дизайне.

С другой стороны, преимущество этих функций по сравнению с другими, более сложными методами взаимодействия, которые мы изучим в дальнейшем — как раз в том, что они очень просты.

Это самый простой способ вывести сообщение или получить информацию от посетителя. Поэтому их используют в тех случаях, когда простота важна, а всякие «красивости» особой роли не играют.

## Оператор if

Оператор if («если») получает условие, в примере выше это year != 2011. Он вычисляет его, и если результат — true, то выполняет команду.

Если нужно выполнить более одной команды — они оформляются блоком кода в фигурных скобках:

if (year != 2011)

{

alert('А вот..');

alert('..и неправильно!');

}

Рекомендуется использовать фигурные скобки всегда, даже когда команда одна. Это улучшает читаемость кода.

## Преобразование к логическому типу

Оператор if вычисляет и преобразует cond к логическому типу.

В логическом контексте число 0, пустая строка "", null и undefined, а также NaN являютсяfalse, 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

...

}

Вычисление условия в проверке if (year != 2011) может быть вынесено в отдельную переменную:

var cond = (year != 2011); // true/false

if (cond)

{

...

}

## Неверное условие, else

Необязательный блок else («иначе») выполняется, если условие неверно:

var year = prompt('Введите год ECMA-262 5.1', '');

if (year == 2011)

{

alert('Да вы знаток!');

}

else

{

alert('А вот и неправильно!'); // любое значение, кроме 2011

}

## Несколько условий, else if

Бывает нужно проверить несколько вариантов условия. Для этого используется блок else if .... Например:

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

if (year < 2011)

{

alert('Это слишком рано..');

}

else if (year > 2011)

{

alert('Это поздновато..');

}

else

{

alert('Да, точно в этом году!');

}

В примере выше JavaScript сначала проверит первое условие, если оно ложно — перейдет ко второму — и так далее, до последнего else.

## || (ИЛИ)

Оператор ИЛИ выглядит как двойной символ вертикальной черты:

result = a || b;

Логическое ИЛИ в классическом программировании работает следующим образом: «еслихотя бы один из аргументов true, то возвращает true, иначе — false».

Получается следующая таблица результатов:

alert( true || true ); // true

alert( false || true ); // true

alert( true || false); // true

alert( false || false); // false

При вычислении ИЛИ в JavaScript можно использовать любые значения. В этом случае они будут интерпретироваться как логические.

Например, число 1 будет воспринято как true, а 0 — как false:

if ( 1 || 0 )

{ // сработает как if( true || false )

alert('верно');

}

Обычно оператор ИЛИ используется в if, чтобы проверить, выполняется ли хотя бы одно из условий, например:

var hour = 9;

if (hour < 10 || hour > 18)

{

alert('Офис до 10 или после 18 закрыт');

}

Можно передать и больше условий:

var hour = 12, isWeekend = true;

if (hour < 10 || hour > 18 || isWeekend)

{

alert('Офис до 10 или после 18 или в выходной закрыт');

}

## Короткий цикл вычислений

JavaScript вычисляет несколько ИЛИ слева направо. При этом, чтобы экономить ресурсы, используется так называемый «короткий цикл вычисления».

Допустим, вычисляются несколько ИЛИ подряд: a || b || c || .... Если первый аргумент — true, то результат заведомо будет true (хотя бы одно из значений — true), и остальные значения игнорируются.

Это особенно заметно, когда выражение, переданное в качестве второго аргумента, имеет сторонний эффект — например, присваивает переменную.

При запуске примера ниже присвоение x не произойдёт:

var x;

true || (x = 1); // просто вычислим ИЛИ, без if

alert(x); // undefined, x не присвоен

А в примере ниже первый аргумент — false, так что ИЛИ попытается вычислить второй, запустив тем самым присваивание:

var x;

false || (x = 1);

alert(x); // 1

## Значение ИЛИ

Итак, как мы видим, оператор ИЛИ вычисляет ровно столько значений, столько необходимо — до первого true.

Оператор ИЛИ возвращает то значение, на котором остановились вычисления.

Примеры:

alert( 1 || 0 ); // 1

alert( true || 'неважно что'); // true

alert( null || 1 ); // 1

alert( undefined || 0 ); // 0

Это используют, в частности, чтобы выбрать первое «истинное» значение из списка:

var undef; // переменная не присвоена, т.е. равна undefined

var zero = 0;

var emptyStr = "";

var msg = "Привет!";

var result = undef || zero || emptyStr || msg || 0;

alert(result) // выведет "Привет!" - первое значение, которое является true

## && (И)

Оператор И пишется как два амперсанда &&:

result = a && b;

В классическом программировании И возвращает true, если оба аргумента истинны, а иначе — false

alert( true && true ); // true

alert( false && true ); // false

alert( true && false); // false

alert( false && false); // false

Пример:

var hour = 12, minute = 30;

if (hour == 12 && minute == 30)

{

alert('Время 12:30');

}

Как и в ИЛИ, допустимы любые значения:

if ( 1 && 0 )

{ // вычислится как true && false

alert('не сработает, т.к. условие ложно');

}

К И применим тот же принцип «короткого цикла вычислений», но немного по-другому, чем к ИЛИ.

Если левый аргумент — false, оператор И возвращает его и заканчивает вычисления. Иначе — вычисляет и возвращает правый аргумент.

Например:

// Первый аргумент - true,

// Поэтому возвращается второй аргумент

alert(1 && 0); // 0

alert(1 && 5); // 5

// Первый аргумент - false,

// Он и возвращается, а второй аргумент игнорируется

alert(null && 5); // null

alert(0 && "не важно"); // 0

Приоритет оператора И && больше, чем ИЛИ ||, т.е. он выполняется раньше.

Поэтому в следующем коде сначала будет вычислено правое И: 1 && 0 = 0, а уже потом — ИЛИ.

alert(5 || 1 && 0); // 5

## Не используйте && вместо if

Оператор && в простых случаев можно использовать вместо if, например:

var x = 1;

(x > 0) && alert('Больше');

Действие в правой части && выполнится только в том случае, если до него дойдут вычисления. То есть, если в левой части будет true.

Получился аналог:

var x = 1;

if (x > 0)

{

alert('Больше');

}

Однако, как правило, if лучше читается и воспринимается. Он более очевиден, поэтому лучше использовать его. Это, впрочем, относится и к другим неочевидным применениям возможностей языка.

## ! (НЕ)

Оператор НЕ — самый простой. Он получает один аргумент. Синтаксис:

var result = !value;

Действия !:

1. Сначала приводит аргумент к логическому типу true/false.

2. Затем возвращает противоположное значение.

Например:

alert( !true ) // false

alert( !0 ) // true

В частности, двойное НЕ используются для преобразования значений к логическому типу:

alert( !!"строка" ) // true

alert( !!null ) // false

## Цикл while

Цикл while имеет вид:

while (условие)

{

// код, тело цикла

}

Пока условие верно — выполняется код из тела цикла.

Например, цикл ниже выводит i пока i < 3:

var i = 0;

while (i < 3)

{

alert(i);

i++;

}

Повторение цикла по-научному называется «итерация». Цикл в примере выше совершает три итерации.

Если бы i++ в коде выше не было, то цикл выполнялся бы (в теории) вечно. На практике, браузер выведет сообщение о «зависшем» скрипте и посетитель его остановит.

Бесконечный цикл можно сделать и проще:

while (true)

{

// ...

}

Условие в скобках интерпретируется как логическое значение, поэтому вместо while (i!=0) обычно пишут while (i):

var i = 3;

while (i)

{ // при i=0 значение в скобках будет false и цикл остановится

alert(i);

i--;

}

## Цикл do..while

Проверку условия можно поставить под телом цикла, используя специальный синтаксис do..while:

do

{

// тело цикла

} while (условие);

Цикл, описанный, таким образом, сначала выполняет тело, а затем проверяет условие.

Например:

var i = 0;

do

{

alert(i);

i++;

} while (i < 3);

Синтаксис do..while редко используется, т.к. обычный while нагляднее — в нём не приходится искать глазами условие и ломать голову, почему оно проверяется именно в конце.

## Цикл for

Чаще всего применяется цикл for. Выглядит он так:

for (начало; условие; шаг)

{

// ... тело цикла ...

}

Например, цикл ниже выводит значения от 0 до 3 (не включая 3):

var i;

for (i=0; i<3; i++)

{

alert(i);

}

• Начало i=0 выполняется при заходе в цикл.

• Условие i<3 проверяется перед каждой итерацией.

• Шаг i++ выполняется после каждой итерации, но перед проверкой условия.

В цикле также можно определить переменную:

for (var i=0; i<3; i++)

{

...

}

Любая часть for может быть пропущена.

Например, можно убрать начало:

var i = 0;

for (; i<3; i++) ...

Можно убрать и шаг:

var i = 0;

for (; i<3; )

{

// цикл превратился в аналог while (i<3)

}

А можно и вообще убрать все, получив бесконечный цикл:

for (;;)

{

// будет выполняться вечно

}

При этом сами точки с запятой ; обязательно должны присутствовать, иначе будет ошибка синтаксиса.

for..in

Существует также специальная конструкция for..in для перебора свойств объекта.

Мы познакомимся с ней позже, когда будем говорить об объектах.

## Выход: break

Выйти из цикла можно не только при проверке условия но и, вообще, в любой момент. Эту возможность обеспечивает директива break.

Например, бесконечный цикл в примере прекратит выполнение при i==5:

var i=0;

while(1)

{

i++;

if (i==5) break;

alert(i);

}

alert('Последняя i = '+ i ); // 5 (\*)

Выполнение продолжится со строки (\*), следующей за циклом.

## Следующая итерация: continue

Директива continue прекращает выполнение текущей итерации цикла. Например, цикл ниже не выводит четные значения:

for (var i = 0; i < 10; i++)

{

if (i % 2 == 0) continue;

alert(i);

}

Для четных i срабатывает continue, выполнение блока прекращается и управление передается на for.

## Ссылки для самых любознательных

1. <http://javascript.ru/>
2. <http://learn.javascript.ru/>
3. <http://www.w3schools.com/js/default.asp>
4. <http://ru.wikipedia.org/wiki/JavaScript>