JavaScript

JavaScript - мультипарадигменный язык программирования. Поддерживает объектно-ориентированный, императивный и функциональный стили. Является реализацией языка ECMAScript.

Автор - Брендан Эйх. Тип исполнения - интерпретация.

Движки (*специализированная программа, обрабатывающая JavaScript, в частности, в браузерах*): SpiderMonkey (первый в истории движок JavaScript), Rhino, V8.

Структурно JavaScript можно представить в виде объединения трёх чётко различимых друг от друга частей:

- ядро (ECMAScript)
- объектная модель браузера (Browser Object Model или BOM)
- объектная модель документа (Document Object Model или DOM)

ECMAScript - это встраиваемый расширяемый не имеющий средств ввода-вывода язык программирования, используемый в качестве основы для построения других скриптовых языков. Стандартизирован международной организацией ECMA в спецификации ECMA-262. Расширения языка: JavaScript, JScript и ActionScript.

JavaScript is case sensitive!

Для выполнения программ, не важно на каком языке, существуют два способа: «компиляция» и «интерпретация».

Компиляция – это когда исходный код программы, при помощи специального инструмента, другой программы, которая называется «компилятор», преобразуется в другой язык, как правило – в машинный код. Этот машинный код затем распространяется и запускается. При этом исходный код программы остаётся у разработчика.

Интерпретация – это когда исходный код программы получает другой инструмент, который называют «интерпретатор», и выполняет его «как есть». При этом распространяется именно сам исходный код (скрипт). Этот подход применяется в браузерах для JavaScript.

Современные интерпретаторы перед выполнением преобразуют JavaScript в машинный код или близко к нему, оптимизируют, а уже затем выполняют. И даже во время выполнения стараются оптимизировать. Поэтому JavaScript работает очень быстро.

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

JavaScript, работающий в одной вкладке, 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 они требуют специального кода на оба документа, которые находятся в разных вкладках или окнах. Без него, из соображений безопасности, залезть из одной вкладки в другую при помощи JavaScript нельзя.

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

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

Благодаря этому один и тот же большой скрипт, содержащий, к примеру, библиотеку функций, может использоваться на разных страницах без полной перезагрузки с сервера.

Если указан атрибут src, то содержимое тега игнорируется. В одном теге script нельзя одновременно подключить внешний скрипт и указать код.

Асинхронные скрипты: defer/async

Браузер загружает и отображает HTML постепенно. Особенно это заметно при медленном интернет-соединении: браузер не ждёт, пока страница загрузится целиком, а показывает ту часть, которую успел загрузить.

Если браузер видит тег <script>, то он по стандарту обязан сначала выполнить его, а потом показать оставшуюся часть страницы.

Такое поведение называют «синхронным». Как правило, оно вполне нормально, но есть важное следствие. Если скрипт – внешний, то пока браузер не выполнит его, он не покажет часть страницы под ним.

Атрибут async

Поддерживается всеми браузерами, кроме IE9-. Скрипт выполняется полностью асинхронно. То есть, при обнаружении <script async src="..."> браузер не останавливает обработку страницы, а спокойно работает дальше. Когда скрипт будет загружен – он выполнится.

Атрибут defer

Поддерживается всеми браузерами, включая самые старые IE. Скрипт также выполняется асинхронно, не заставляет ждать страницу, но есть два отличия от async.

Первое – браузер гарантирует, что относительный порядок скриптов с defer будет сохранён. Поэтому атрибут defer используют в тех случаях, когда второй скрипт 2.js зависит от первого 1.js, к примеру – использует что-то, описанное первым скриптом.

Второе отличие – скрипт с defer сработает, когда весь HTML-документ будет обработан браузером. Например, если документ достаточно большой, то скрипт async.js выполнится, как только загрузится – возможно, до того, как весь документ готов. А defer.js подождёт готовности всего документа. Это бывает удобно, когда мы в скрипте хотим работать с документом, и должны быть уверены, что он полностью получен.

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

Многострочные комментарии начинаются слешем-звездочкой «/*» и заканчиваются звездочкой-слэшем «*/».

Вложенные комментарии не поддерживаются!

Современный стандарт, "use strict"

Очень долго язык JavaScript развивался без потери совместимости. Новые возможности добавлялись в язык, но старые – никогда не менялись, чтобы не «сломать» уже существующие HTML/JS-страницы с их использованием. Однако, это привело к тому, что любая ошибка в дизайне языка становилась «вмороженной» в него навсегда. Так было до появления стандарта ECMAScript 5 (ES5), который одновременно добавил новые возможности и внёс в язык ряд исправлений, которые могут привести к тому, что старый код, который был написан до его появления, перестанет работать.

Чтобы этого не случилось, решили, что по умолчанию эти опасные изменения будут выключены, и код будет работать по-старому. А для того, чтобы перевести код в режим полного соответствия современному стандарту, нужно указать специальную директиву use strict.

Директива выглядит как строка "use strict"; или 'use strict'; и ставится в начале скрипта.

Не существует директивы no use strict или подобной, которая возвращает в старый режим.

use strict также можно указывать в начале функций, тогда строгий режим будет действовать только внутри функции.

ES5-shim

Браузер IE8 поддерживает только совсем старую версию стандарта JavaScript, а именно ES3.

К счастью, многие возможности современного стандарта можно добавить в этот браузер, подключив библиотеку ES5 shim [https://github.com/es-shims/es5-shim], а именно – скрипты es5-shim.js и es5-sham.js из неё.

Переменные

Переменная состоит из имени и выделенной области памяти, которая ему соответствует. Для объявления или, другими словами, создания переменной используется ключевое слово var.

var message;

message = ‘text’;

var message = ‘text’;

var name = ‘John’, age = 25;

На имя переменной в JavaScript наложены всего два ограничения:

1. Имя может состоять из: букв, цифр, символов $ и _
2. Первый символ не должен быть цифрой.

Существует список зарезервированных слов, которые нельзя использовать для переменных, так как они используются самим языком, например: var, class, return, export и др.

В старом стандарте JavaScript разрешалось создавать переменную и без var, просто присвоив ей значение:

number = 5; // переменная будет создана, если ее не было

В режиме "use strict" так делать уже нельзя.

Переменные из нескольких слов пишутся вместеВотТак.

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

Константы

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

var COLOR_RED = "#F00";

var COLOR_GREEN = "#0F0";

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

Шесть типов данных, typeof

Число «number»

Единый тип число используется как для целых, так и для дробных чисел. Существуют специальные числовые значения Infinity (бесконечность) и NaN (ошибка вычислений). Например, бесконечность Infinity получается при делении на ноль. Ошибка вычислений NaN будет результатом некорректной математической операции.

var number = 5;

Строка «string»

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 C это char. В JavaScript есть только тип «строка» string.

var string = ‘Hello, World!’;

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.

var checked = true;

Специальное значение «null»

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.

Специальное значение «undefined»

Значение undefined, как и null, образует свой собственный тип, состоящий из одного этого значения. Оно имеет смысл «значение не присвоено».

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

var x;

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

var x = undefined;

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

Объекты «object»

Первые 5 типов (number, string, boolean, null, undefined) называют «примитивными».

Особняком стоит шестой тип: «объекты».

Объявляются объекты при помощи фигурных скобок {...}, например:

var user = { name: ‘John’ };

Оператор typeof

Оператор typeof возвращает тип аргумента.

У него есть два синтаксиса: со скобками и без:

1. Синтаксис оператора: typeof x
2. Синтаксис функции: typeof(x)

Результатом typeof является строка, содержащая тип.

Основные операторы

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 операнду. 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.

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! Это приведение к строке – особенность исключительно бинарного оператора "+". Остальные арифметические операторы работают только с числами и всегда приводят аргументы к числу.

Результат a % b – это остаток от деления a на b.

Инкремент ++ увеличивает на 1.

Декремент -- уменьшает на 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. **Постфиксная форма i++ отличается от префиксной ++i тем, что возвращает старое значение, бывшее до увеличения.**

Строки сравниваются побуквенно.

При сравнении значений разных типов, используется числовое преобразование.

В обычном операторе == есть «проблема» – он не может отличить 0 от false. Это естественное следствие того, что операнды разных типов преобразовались к числу. Пустая строка, как и false, при преобразовании к числу дают 0.

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

Интуитивно кажется, что null/undefined эквивалентны нулю, но это не так.

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.

Взаимодействие с пользователем: alert, prompt, confirm

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

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.

Вызов prompt возвращает то, что ввёл посетитель – строку или специальное значение null, если ввод отменён.

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

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

Условные операторы: if, '?'

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

Оператор if («если») получает условие. Он вычисляет его, и если результат – true, то выполняет команду. Рекомендуется использовать фигурные скобки всегда, даже когда команда одна.

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

- Число 0, пустая строка "", null и undefined, а также NaN являются false,
- Остальные значения – true.

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

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

Иногда нужно в зависимости от условия присвоить переменную. Оператор вопросительный знак '?' позволяет делать это короче и проще.

условие ? значение1 : значение2

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».

Для операций над логическими значениями в JavaScript есть || (ИЛИ), && (И) и ! (НЕ).

Преобразование типов для примитивов

Всего есть три преобразования:

1. Строковое преобразование.
2. Численное преобразование.
3. Преобразование к логическому значению.

Строковое преобразование происходит, когда требуется представление чего-либо в виде строки. Например, его производит функция alert. Можно также осуществить преобразование явным вызовом String(val).

Численное преобразование происходит в математических функциях и выражениях, а также при сравнении данных различных типов (кроме сравнений ===, !==). Для преобразования к числу в явном виде можно вызвать Number(val), либо, что короче, поставить перед выражением унарный плюс "+".

Преобразование к true/false происходит в логическом контексте, таком как if(value), и при применении логических операторов. Все значения, которые интуитивно «пусты», становятся false. Их несколько: 0, пустая строка, null, undefined и NaN. Остальное, в том числе и любые объекты – true.

Для явного преобразования используется двойное логическое отрицание !!value или вызов Boolean(value).

В отличие от многих языков программирования (например PHP), "0" в JavaScript является true, как и строка из пробелов.

Цикл while

Цикл while имеет вид:

```
while (условие) {
	// код, тело цикла
}
```

Пока условие верно – выполняется код из тела цикла.

Повторение цикла по-научному называется «итерация».

Цикл do…while

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

```
do {
    // тело цикла
} while (условие);
```

Цикл for

```
for (начало; условие; шаг) {
    // ... тело цикла ...
}
```

Цикл выполняется так:

1. Начало: i=0 выполняется один-единственный раз, при заходе в цикл.
2. Условие: i<3 проверяется перед каждой итерацией и при входе в цикл, если оно нарушено, то происходит выход.
3. Тело: alert(i).
4. Шаг: i++ выполняется после тела на каждой итерации, но перед проверкой условия.
5. Идти на шаг 2.

В цикле также можно определить переменную:

```
for (var i = 0; i < 3; i++) {
    alert(i); // 0, 1, 2
}
```

Эта переменная будет видна и за границами цикла, в частности, после окончания цикла i станет равно 3.

Любая часть for может быть пропущена. Например, можно убрать начало. Цикл в примере ниже полностью идентичен приведённому выше:

```
var i = 0;
for (; i < 3; i++) {
   alert( i ); // 0, 1, 2
}
```

Можно убрать и шаг. А можно и вообще убрать всё, получив бесконечный цикл: for (;;) { // будет выполняться вечно }. Существует также специальная конструкция for..in для перебора свойств объекта.

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

Директива **continue** прекращает выполнение текущей итерации цикла.

Нельзя использовать break/continue справа от оператора „?“.

Синтаксические конструкции, которые не возвращают значений, нельзя использовать в операторе '?'.

Конструкция **switch** заменяет собой сразу несколько if.

```
switch(x) {
    case 'value1': // if (x === 'value1')
        ...
        [break]
    case 'value2': // if (x === 'value2')
        ...
        [break]
    default:
        ...
        [break]

}
```

Несколько значений case можно группировать.

Функции

```
function showMessage() {
   alert('Привет всем присутствующим!');
}
```

Вначале идет ключевое слово function, после него имя функции, затем список параметров в скобках (в примере выше он пустой) и тело функции – код, который выполняется при её вызове.

Функция может содержать локальные переменные, объявленные через var. Такие переменные видны только внутри функции. Блоки if/else, switch, for, while, do..while не влияют на область видимости переменных. Неважно, где именно в функции и сколько раз объявляется переменная. Любое объявление срабатывает один раз и распространяется на всю функцию.

Функция может обратиться ко внешней переменной. Доступ возможен не только на чтение, но и на запись. При этом, так как переменная внешняя, то изменения будут видны и снаружи функции.

Переменные, объявленные на уровне всего скрипта, называют «глобальными переменными».

Делайте глобальными только те переменные, которые действительно имеют общее значение для вашего проекта, а нужные для решения конкретной задачи – пусть будут локальными в соответствующей функции.

При вызове функции ей можно передать данные, которые та использует по своему усмотрению. **Параметры копируются в локальные переменные функции.**

Функцию можно вызвать с любым количеством аргументов. Если параметр не передан при вызове – он считается равным undefined.

Для возврата значения используется директива return.

Имя функции следует тем же правилам, что и имя переменной. Основное отличие – оно должно быть глаголом, т.к. функция – это действие. Как правило, используются глагольные префиксы, обозначающие общий характер действия, после которых следует уточнение.

В JavaScript функция является значением, таким же как строка или число.

```
function sayHi() {
    alert( "Привет" );
}

alert( sayHi ); // выведет код функции
```

Обратим внимание на то, что в последней строке после sayHi нет скобок. То есть, функция не вызывается, а просто выводится на экран.

Функцию можно скопировать в другую переменную.

Существует альтернативный синтаксис для объявления функции, который ещё более наглядно показывает, что функция – это всего лишь разновидность значения переменной. Он называется «**Function Expression**» (функциональное выражение) и выглядит так:

```
var f = function(параметры) {
    // тело функции
};
```

Например:

```
var sayHi = function(name) {
     alert(“Hello, ” + name);
}

sayHi(“John”);     // Hello, John
```

«Классическое» объявление функции, о котором мы говорили до этого, вида function имя(параметры) {...}, называется в спецификации языка «Function Declaration».

Function Declaration – функция, объявленная в основном потоке кода.

Function Expression – объявление функции в контексте какого-либо выражения, например присваивания.

Основное отличие между ними: функции, объявленные как Function Declaration, создаются интерпретатором до выполнения кода. Поэтому их можно вызвать до объявления.

Функциональное выражение, которое не записывается в переменную, называют **анонимной функцией**.

Функции в JavaScript являются значениями. Их можно присваивать, передавать, создавать в любом месте кода.

Рекурсия, стек

В теле функции могут быть вызваны другие функции для выполнения подзадач. Частный случай подвызова – когда функция вызывает сама себя. Это называется рекурсией.

У каждого вызова функции есть свой «контекст выполнения» (execution context).

Контекст выполнения – это служебная информация, которая соответствует текущему запуску функции. Она включает в себя локальные переменные функции и конкретное место в коде, на котором находится интерпретатор.

При любом вложенном вызове JavaScript запоминает текущий контекст выполнения в специальной внутренней структуре данных – «стеке контекстов».

Затем интерпретатор приступает к выполнению вложенного вызова.

Для нового вызова создаётся свой контекст выполнения, и управление переходит в него, а когда он завершён – старый контекст достаётся из стека и выполнение внешней функции возобновляется.

Любая рекурсия может быть переделана в цикл. Как правило, вариант с циклом будет эффективнее.

Специально для работы с рекурсией в JavaScript существует особое расширение функциональных выражений, которое называется «Named Function Expression» (сокращённо NFE) или, по-русски, «именованное функциональное выражение».

Обычное функциональное выражение:

var f = function(...) { /* тело функции */ };

Именованное с именем sayHi:

var f = function sayHi(...) { /* тело функции */ };

Имя функционального выражения (sayHi) имеет особый смысл. Оно доступно только изнутри самой функции (f).

Деление на ноль, Infinity

Чем меньше делитель, тем больше результат. При делении на очень-очень маленькое число должно получиться очень большое. В математическом анализе это описывается через пределы, и если подразумевать предел, то в качестве результата деления на 0 мы получаем «бесконечность», которая обозначается символом ∞ (в JavaScript Infinity).

```
alert( 1 / 0 );    // Infinity
```

Infinity – особенное численное значение, которое ведет себя в точности как математическая бесконечность ∞.

Infinity больше любого числа.

Бесконечность можно присвоить и в явном виде: var x = Infinity. Бывает и минус бесконечность –Infinity.

NaN

Если математическая операция не может быть совершена, то возвращается специальное значение NaN (Not-A-Number).

Например, деление 0/0 в математическом смысле неопределено, поэтому его результат NaN.

Значение NaN используется для обозначения математической ошибки и обладает следующими свойствами:

- Значение NaN – единственное в своем роде, которое не равно ничему, включая себя.
- Значение NaN можно проверить специальной функцией isNaN(n), которая преобразует аргумент к числу и возвращает true, если получилось NaN, и false – для любого другого значения.
- Значение NaN «прилипчиво». Любая операция с NaN возвращает NaN.

Никакие математические операции в JavaScript не могут привести к ошибке или «обрушить» программу. В худшем случае результат будет NaN.

Функция parseInt и ее аналог parseFloat преобразуют строку символ за символом, пока это возможно.

Для проверки строки на число можно использовать функцию isNaN(str).

Одна из самых частых операций с числом – округление. В JavaScript существуют целых 3 функции для этого.

Math.floor() - Округляет вниз;

Math.ceil() - Округляет вверх;

Math.round() - Округляет до ближайшего целого.

Объекты как ассоциативные массивы

Объекты в JavaScript сочетают в себе два важных функционала.

Первый – это ассоциативный массив: структура, пригодная для хранения любых данных.

Второй – языковые возможности для объектно-ориентированного программирования.

Ассоциативный массив – структура данных, в которой можно хранить любые данные в формате ключ-значение.

Пустой объект («пустой шкаф») может быть создан одним из двух синтаксисов:

```
o = new Object();
o = {};
```

Объект может содержать в себе любые значения, которые называются свойствами объекта. Доступ к свойствам осуществляется по имени свойства (иногда говорят «по ключу»).

Основные операции с объектами – это создание, получение и удаление свойств.

Для обращения к свойствам используется запись «через точку», вида объект.свойство.

```
person.name = 'Вася';
person.age = 25;
```

Чтобы прочитать их – также обратимся через точку.

Удаление осуществляется оператором delete:

```
delete person.age;
```

Иногда бывает нужно проверить, есть ли в объекте свойство с определенным ключом. Для этого есть особый оператор: "in".

Его синтаксис: "prop" in obj, причем имя свойства – в виде строки, например:

```
if ("name" in person) {
    alert( "Свойство name существует!" );
}
```

Впрочем, чаще используется другой способ – сравнение значения с undefined.

Дело в том, что в JavaScript можно обратиться к любому свойству объекта, даже если его нет. Ошибки не будет. Но если свойство не существует, то вернется специальное значение undefined.

Есть два средства для проверки наличия свойства в объекте: первое – оператор in, второе – получить его и сравнить с undefined. Разница: Технически возможно, что свойство есть, а его значением является undefined. В таком случае метод сравнения не сработает. А оператор in вернет правильный результат.

Существует альтернативный синтаксис работы со свойствами, использующий квадратные скобки объект['свойство']:

```
var person = {};
person['name'] = 'Вася';    // то же что и person.name = 'Вася'
```

Записи person['name'] и person.name идентичны, но квадратные скобки позволяют использовать в качестве имени свойства любую строку:

```
var person = {};
person['любимый стиль музыки'] = 'Джаз';
```

Квадратные скобки также позволяют обратиться к свойству, имя которого хранится в переменной:

```
var person = {}, key = 'name';
person.name = 'John';

console.log(person.key);  // undefined
console.log(person[key]); // John
```

Объект можно заполнить значениями при создании, указав их в фигурных скобках: { ключ1: значение1, ключ2: значение2, ... }. Такой синтаксис называется литеральным (англ. literal).

Следующие два фрагмента кода создают одинаковый объект:

```
var menuSetup = {
    width: 300,
    height: 200,
    title: "Menu"
};
```

```
var menuSetup = {};
menuSetup.width = 300;
menuSetup.height = 200;
menuSetup.title = 'Menu';
```

Названия свойств можно перечислять как в кавычках, так и без, если они удовлетворяют ограничениям для имён переменных.

В качестве значения можно тут же указать и другой объект.

Для перебора всех свойств из объекта используется цикл по свойствам for..in

```
for (key in obj) {
     /* ... делать что-то с obj[key] ... */
}
```

При этом for..in последовательно переберёт свойства объекта obj, имя каждого свойства будет записано в key и вызвано тело цикла.

```
var menu = { width: 300, height: 200, title: "Menu" };

for (var key in menu) {
    alert( "Ключ: " + key + " значение: " + menu[key] );
}
```

Фундаментальным отличием объектов от примитивов, является их хранение и копирование «по ссылке». Обычные значения: строки, числа, булевы значения, null/undefined при присваивании переменных копируются целиком или, как говорят, «по значению».

Копирование по значению.

```
var message = "Привет";
var phrase = message;
```

В результате такого копирования получились две полностью независимые переменные, в каждой из которых хранится значение "Привет".

Копирование по ссылке

С объектами – всё не так. В переменной, которой присвоен объект, хранится не сам объект, а «адрес его места в памяти», иными словами – «ссылка» на него.

Вот как выглядит переменная, которой присвоен объект:

```
var user = { name: "Вася" };
```

При копировании переменной с объектом – копируется эта ссылка, а объект по-прежнему остается в единственном экземпляре.

Например:

```
var user = { name: "Вася" };   // в переменной – ссылка
var admin = user;   // скопировали ссылку
```

Получили две переменные, в которых находятся ссылки на один и тот же объект.

Так как объект всего один, то изменения через любую переменную видны в других переменных:

```
var user = { name: 'Вася' };
var admin = user;

admin.name = 'Петя';    // поменяли данные через admin
alert(user.name);    // 'Петя', изменения видны в user
```

Синтаксис для создания нового массива – квадратные скобки со списком элементов внутри.

Пустой массив:

```
var arr = [];
```

Массив fruits с тремя элементами:

```
var fruits = ["Яблоко", "Апельсин", "Слива"];
```

Элементы нумеруются, начиная с нуля.

Чтобы получить нужный элемент из массива – указывается его номер в квадратных скобках:

```
var fruits = ["Яблоко", "Апельсин", "Слива"];

alert( fruits[0] ); // Яблоко
alert( fruits[1] ); // Апельсин
alert( fruits[2] ); // Слива
```

Элемент можно всегда заменить:

```
fruits[2] = 'Груша';  // теперь ["Яблоко", "Апельсин", "Груша"]
```

…Или добавить:

```
fruits[3] = 'Лимон'; // теперь ["Яблоко", "Апельсин", "Груша", "Лимон"]
```

Общее число элементов, хранимых в массиве, содержится в его свойстве length.

Через alert можно вывести и массив целиком. При этом его элементы будут перечислены через запятую:

```
var fruits = ["Яблоко", "Апельсин", "Груша"];
alert( fruits ); // Яблоко,Апельсин,Груша
```

В массиве может храниться любое число элементов любого типа.

В том числе, строки, числа, объекты, вот например:

```
// микс значений
var arr = [ 1, 'Имя', { name: 'Петя' }, true ];

// получить объект из массива и тут же -- его свойство
alert( arr[2].name ); // Петя
```

Методы pop/push, shift/unshift

Одно из применений массива – это **очередь**. В классическом программировании так называют упорядоченную коллекцию элементов, такую что элементы добавляются в конец, а обрабатываются – с начала.

Очень близка к очереди еще одна структура данных: **стек**. Это такая коллекция элементов, в которой новые элементы добавляются в конец и берутся с конца.

pop()

Удаляет последний элемент из массива и возвращает его:

```
var fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.pop() );    // удалили "Груша"
alert( fruits );     // Яблоко, Апельсин
```

push()

Добавляет элемент в конец массива:

```
var fruits = ["Яблоко", "Апельсин"];

fruits.push("Груша");
alert( fruits );   // Яблоко, Апельсин, Груша
```

shift()

Удаляет из массива первый элемент и возвращает его:

```
var fruits = ["Яблоко", "Апельсин", "Груша"];

alert( fruits.shift() );      // удалили Яблоко
alert( fruits );      // Апельсин, Груша
```

unshift()

Добавляет элемент в начало массива:

```
var fruits = ["Апельсин", "Груша"];

fruits.unshift('Яблоко');
alert( fruits );    // Яблоко, Апельсин, Груша
```

Методы push и unshift могут добавлять сразу по несколько элементов через запятую.

Массив – это объект, где в качестве ключей выбраны цифры, с дополнительными методами и свойством length.

Методы push/pop выполняются быстро, а shift/unshift – медленно.

Длина length – не количество элементов массива, а последний индекс + 1.

```
var arr = [];
arr[1000] = true;

alert(arr.length);   // 1001
```

Существует еще один синтаксис для создания массива:

```
var arr = new Array("Яблоко", "Груша", "и т.п.");
```

Обычно new Array(элементы, ...) создаёт массив из данных элементов, но если у него один аргумент-число new Array(число), то он создает массив без элементов, но с заданной длиной. Получившийся массив ведёт себя так, как будто его элементы равны undefined.

Массивы в JavaScript могут содержать в качестве элементов другие массивы. Это можно использовать для создания многомерных массивов, например матриц.

```
var matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
];

alert( matrix[1][1] );     // центральный элемент - 5
```

---

join/split – для преобразования строки в массив и обратно.

slice – копирует участок массива.

sort – для сортировки массива. Если не передать функцию сравнения – сортирует элементы как строки.

reverse – меняет порядок элементов на обратный.

concat – объединяет массивы.

indexOf/lastIndexOf – возвращают позицию элемента в массиве.

Массив: перебирающие методы

forEach

Метод «arr.forEach(callback[, thisArg])» используется для перебора массива. Он для каждого элемента массива вызывает функцию callback. Этой функции он передаёт три параметра callback(item, i, arr):

item – очередной элемент массива.
i – его номер.
arr – массив, который перебирается.

Например:

```
var arr = ["Яблоко", "Апельсин", "Груша"];

arr.forEach(function(item, i, arr) {
     alert( i + ": " + item + " (массив:" + arr + ")" );
});
```

forEach – для перебора массива.
filter – для фильтрации массива.
every/some – для проверки массива.
map – для трансформации массива в массив.
reduce/reduceRight – для прохода по массиву с вычислением значения.

В JavaScript любая функция может быть вызвана с произвольным количеством аргументов.

Например:

```
function go(a,b) {
   alert("a="+a+", b="+b);
}

go(1);         // a=1, b=undefined
go(1,2);       // a=1, b=2
go(1,2,3);    // a=1, b=2, третий аргумент не вызовет ошибку
```

В JavaScript нет «перегрузки» функций

В некоторых языках программист может создать две функции с одинаковым именем, но разным набором аргументов, а при вызове интерпретатор сам выберет нужную:

```
function log(a) { ... }

function log(a, b, c) { ... }

log(a);          // вызовется первая функция
log(a, b, c);   // вызовется вторая функция
```

Это называется «полиморфизмом функций» или «перегрузкой функций». В JavaScript ничего подобного нет.

Может быть только одна функция с именем log, которая вызывается с любыми аргументами.

В примере выше второе объявление log просто переопределит первое.

Полный список аргументов, с которыми вызвана функция, доступен через arguments.

Это псевдомассив, то есть объект, который похож на массив, в нём есть нумерованные свойства и length, но методов массива у него нет.

Для работы с датой и временем в JavaScript используются объекты Date. Для создания нового объекта типа Date используется:

new Date()

Создает объект Date с текущей датой и временем:

```
var now = new Date();
alert( now );    // Tue Nov 14 2017 12:01:57 GMT+0400 (Азербайджанское время (зима))
```

Для доступа к компонентам даты-времени объекта Date используются следующие методы:

getFullYear() - Получить год (из 4 цифр)
getMonth() - Получить месяц, от 0 до 11.
getDate() - Получить число месяца, от 1 до 31.
getDay() - Получить день недели. Получить номер дня в неделе. Неделя в JavaScript начинается с воскресенья, так что результат будет числом от 0(воскресенье) до 6(суббота).
getHours(), getMinutes(), getSeconds(), getMilliseconds()

Все методы, указанные выше, возвращают результат для местной временной зоны.

Следующие методы позволяют устанавливать компоненты даты и времени:

setFullYear(year [, month, date])
setMonth(month [, date])
setDate(date)
setHours(hour [, min, sec, ms])
setMinutes(min [, sec, ms])
setSeconds(sec [, ms])
setMilliseconds(ms)
setTime(milliseconds)

Когда объект Date используется в числовом контексте, он преобразуется в количество миллисекунд.

alert(+new Date) // 1510647211601

Важный побочный эффект: даты можно вычитать, результат вычитания объектов Date – их временная разница, в миллисекундах.

Дата и время представлены в JavaScript одним объектом: Date.
Отсчёт месяцев начинается с нуля.
Отсчёт дней недели (для getDay()) тоже начинается с нуля (и это воскресенье).
Объект Date удобен тем, что автокорректируется.
При преобразовании к числу объект Date даёт количество миллисекунд, прошедших с 1 января 1970 UTC. Побочное следствие – даты можно вычитать, результатом будет разница в миллисекундах.

JavaScript

Инициализация функции (немедленный вызов):

```
var plus = function(a,b) {
     return alert(a+b);
} (2,2);
```

---

return – несколько параметров:

```
function plus(a, b) {
    return(
         alert(a+b),  // 4
         alert(this),  // [object Window]
         alert(arguments)  // [object Arguments]
    )
}

plus(2, 2);
```

Функции как объекты:

```
var calc = {
    status: ‘Awesome’,
    plus: function (a, b) {
        return(
            alert(this),    // object Object
            alert(a+b),    // 4
            alert(arguments),    // object Arguments
            alert(this.status)     // Awesome
        )
    }
}

calc.plus(2, 2);
```

Вызываем экземпляры с помощью конструктора:

```
var Dog = function () {
    var name, breed;
}

firstDog = new Dog;
firstDog.name = ‘Rover’;
firstDog.breed = ‘Doberman’;

secondDog = new Dog;
secondDog.name = ‘Fluffy’;
secondDog.breed = ‘Poodle’;

console.log(firstDog.name);
console.log(secondDog.name);
```

Прототип:

```
var speak = function (saywhat) {
     console.log(saywhat);   // woof
}

var Dog = function () {
     var name, breed;
}

Dog.prototype.speak = speak;

firstDog = new Dog;
firstDog.name = ‘Rover’;
firstDog.breed = ‘Doberman’;
firstDog.speak(‘woof’);
```

Глобальными называют переменные и функции, которые не находятся внутри какой-то функции. То есть, иными словами, если переменная или функция не находятся внутри конструкции function, то они – «глобальные».

В JavaScript 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.

Присваивая или читая глобальную переменную, мы, фактически, работаем со свойствами window:

```
var a = 5;    // объявление var создаёт свойство window.a
alert( window.a );  // 5
```

Создать переменную можно и явным присваиванием в window:

```
window.a = 5;
alert( a );   // 5
```

Порядок инициализации

Выполнение скрипта происходит в две фазы:

1. На первой фазе происходит инициализация, подготовка к запуску.

   Во время инициализации скрипт сканируется на предмет объявления функций вида Function Declaration, а затем – на предмет объявления переменных var. Каждое такое объявление добавляется в window.

   Функции, объявленные как Function Declaration, создаются сразу работающими, а переменные – равными undefined.
2. На второй фазе – собственно, выполнение.

   Присваивание (=) значений переменных происходит, когда поток выполнения доходит до соответствующей строчки кода, до этого они undefined.

Конструкции for, if... не влияют на видимость переменных.
Все var будут обработаны один раз, на фазе инициализации.

Лексическое окружение

Все переменные внутри функции – это свойства специального внутреннего объекта LexicalEnvironment, который создаётся при её запуске. Мы будем называть этот объект «лексическое окружение» или просто «объект переменных». При запуске функция создает объект LexicalEnvironment, записывает туда аргументы, функции и переменные. Процесс инициализации выполняется в том же порядке, что и для глобального объекта, который, вообще говоря, является частным случаем лексического окружения. В отличие от window, объект LexicalEnvironment является внутренним, он скрыт от прямого доступа.

Доступ ко внешним переменным

Из функции мы можем обратиться не только к локальной переменной, но и к внешней:

```
var userName = "Вася";

function sayHi() {
    alert( userName );   // "Вася"
}
```

Интерпретатор, при доступе к переменной, сначала пытается найти переменную в текущем LexicalEnvironment, а затем, если её нет – ищет во внешнем объекте переменных. В данном случае им является window.

Такой порядок поиска возможен благодаря тому, что ссылка на внешний объект переменных хранится в специальном внутреннем свойстве функции, которое называется [[Scope]]. Это свойство закрыто от прямого доступа. При создании функция получает скрытое свойство [[Scope]], которое ссылается на лексическое окружение, в котором она была создана.

- Каждая функция при создании получает ссылку [[Scope]] на объект с переменными, в контексте которого была создана.
- При запуске функции создаётся новый объект с переменными LexicalEnvironment. Он получает ссылку на внешний объект переменных из [[Scope]].
- При поиске переменных он осуществляется сначала в текущем объекте переменных, а потом – по этой ссылке.

Всегда текущее значение

Значение переменной из внешней области берётся всегда текущее. Оно может быть уже не то, что было на момент создания функции. Например, в коде ниже функция sayHi берёт phrase из внешней области:

```
var phrase = 'Привет';

function sayHi(name) {
    alert(phrase + ', ' + name);
}

sayHi('Вася');  // Привет, Вася (*)

phrase = 'Пока';

sayHi('Вася');  // Пока, Вася (**)
```

На момент первого запуска (*), переменная phrase имела значение 'Привет', а ко второму (**) изменила его на 'Пока'. Это естественно, ведь для доступа к внешней переменной функция по ссылке [[Scope]] обращается во внешний объект переменных и берёт то значение, которое там есть на момент обращения.

Вложенные функции

Внутри функции можно объявлять не только локальные переменные, но и другие функции. Вложенные функции получают [[Scope]] так же, как и глобальные.

Свойства функции

Функция в JavaScript является объектом, поэтому можно присваивать свойства прямо к ней, вот так:

```
function f() {}

f.test = 5;

alert( f.test );
```

Свойства функции не стоит путать с переменными и параметрами. Они совершенно никак не связаны. Переменные доступны только внутри функции, они создаются в процессе её выполнения. Это – использование функции «как функции». А свойство у функции – доступно отовсюду и всегда. Это – использование функции «как объекта».

Замыкание – это функция вместе со всеми внешними переменными, которые ей доступны. То есть, замыкание – это функция + внешние переменные.

Обычно, говоря «замыкание функции», подразумевают не саму эту функцию, а именно внешние переменные.

Иногда говорят «переменная берётся из замыкания». Это означает – из внешнего объекта переменных.

«Понимать замыкания» в JavaScript означает понимать следующие вещи:

- Все переменные и параметры функций являются свойствами объекта переменных LexicalEnvironment. Каждый запуск функции создает новый такой объект. На верхнем уровне им является «глобальный объект», в браузере – window.
- При создании функция получает системное свойство [[Scope]], которое ссылается на LexicalEnvironment, в котором она была создана.
- При вызове функции, куда бы её ни передали в коде – она будет искать переменные сначала у себя, а затем во внешних LexicalEnvironment с места своего «рождения».

При создании функции с использованием new Function, её свойство [[Scope]] ссылается не на текущий LexicalEnvironment, а на window.

Устаревшая конструкция "with"

Конструкция with позволяет использовать в качестве области видимости для переменных произвольный объект. В современном JavaScript от этой конструкции отказались. С use strict она не работает.

```
with(obj) {
  // код
}
```

Любое обращение к переменной внутри with сначала ищет её среди свойств obj, а только потом – вне with.

Управление памятью в JavaScript

Управление памятью в JavaScript обычно происходит незаметно. Мы создаём примитивы, объекты, функции… Всё это занимает память. Что происходит с объектом, когда он становится «не нужен»? Возможно ли «переполнение» памяти? Для ответа на эти вопросы – залезем «под капот» интерпретатора.

Главной концепцией управления памятью в JavaScript является принцип достижимости (англ. reachability).

1. Определённое множество значений считается достижимым изначально, в частности:

   Значения, ссылки на которые содержатся в стеке вызова, то есть – все локальные переменные и параметры функций, которые в настоящий момент выполняются или находятся в ожидании окончания вложенного вызова.

   Все глобальные переменные.

   Эти значения гарантированно хранятся в памяти. Мы будем называть их корнями.
2. Любое другое значение сохраняется в памяти лишь до тех пор, пока доступно из корня по ссылке или цепочке ссылок.

Для очистки памяти от недостижимых значений в браузерах используется автоматический Сборщик мусора (англ. Garbage collection, GC), встроенный в интерпретатор, который наблюдает за объектами и время от времени удаляет недостижимые.

Самая простая ситуация здесь с примитивами. При присвоении они копируются целиком, ссылок на них не создаётся, так что если в переменной была одна строка, а её заменили на другую, то предыдущую можно смело выбросить. Именно объекты требуют специального «сборщика мусора», который наблюдает за ссылками, так как на один объект может быть много ссылок из разных переменных и, при перезаписи одной из них, объект может быть всё ещё доступен из другой.

JavaScript

Методы у объектов

При объявлении объекта можно указать свойство-функцию. Свойства-функции называют «методами» объектов.

Доступ к объекту через this

Для полноценной работы метод должен иметь доступ к данным объекта. В частности, вызов user.sayHi() может захотеть вывести имя пользователя.

Для доступа к текущему объекту из метода используется ключевое слово this.

```
var user = {
    name: 'Василий',
    sayHi: function() {
        alert( this.name );
    }
};

user.sayHi();  // Василий
```

Здесь при выполнении функции user.sayHi() в this будет храниться ссылка на текущий объект user. Использование this гарантирует, что функция работает именно с тем объектом, в контексте которого вызвана.

Через this метод может не только обратиться к любому свойству объекта, но и передать куда-то ссылку на сам объект целиком:

```
var user = {
    name: 'Василий',
    sayHi: function() {
        showName(this);   // передать текущий объект в showName
    }
};

function showName(namedObj) {
    alert( namedObj.name );
}

user.sayHi();    // Василий
```

Бывают операции, при которых объект должен быть преобразован в примитив. Например:

- Строковое преобразование – если объект выводится через alert(obj).
- Численное преобразование – при арифметических операциях, сравнении с примитивом.
- Логическое преобразование – при if(obj) и других логических операциях.

Проще всего – с **логическим преобразованием**.

Любой объект в логическом контексте – true, даже если это пустой массив [] или объект {}.

Строковое преобразование проще всего увидеть, если вывести объект при помощи alert:

```
var user = { firstName: 'Василий' };

alert( user );    // [object Object]
```

Как видно, содержимое объекта не вывелось. Это потому, что стандартным строковым представлением пользовательского объекта является строка "[object Object]".

Если в объекте присутствует метод toString, который возвращает примитив, то он используется для преобразования:

```
var user = {
    firstName: 'Василий',
    toString: function() {
        return 'Пользователь ' + this.firstName;
    }
};

alert( user );    // Пользователь Василий
```

Результатом toString может быть любой примитив

Для **численного преобразования** объекта используется метод valueOf, а если его нет – то toString.

У большинства объектов нет valueOf

У большинства встроенных объектов такого valueOf нет, поэтому численное и строковое преобразования для них работают одинаково. Исключением является объект Date, который поддерживает оба типа преобразований:

```
alert( new Date() );  // toString: Дата в виде читаемой строки
alert( +new Date() ); // valueOf: кол-во миллисекунд, прошедших с 01.01.1970
```

В логическом контексте объект – всегда true.

При строковом преобразовании объекта используется его метод toString. Он должен возвращать примитивное значение, причём не обязательно именно строку.

Для численного преобразования используется метод valueOf, который также может возвратить любое примитивное значение. У большинства объектов valueOf не работает (возвращает сам объект и потому игнорируется), при этом для численного преобразования используется toString.

Создание объектов через "new"

Обычный синтаксис {...} позволяет создать один объект. Но зачастую нужно создать много однотипных объектов. Для этого используют «функции-конструкторы», запуская их при помощи специального оператора new.

Конструктором становится любая функция, вызванная через new:

```
function Animal(name) {
    this.name = name;
    this.canWalk = true;
}

var animal = new Animal("ёжик");
```

Заметим, что, технически, любая функция может быть использована как конструктор. То есть, любую функцию можно вызвать при помощи new. Как-то особым образом указывать, что она – конструктор – не надо. Но, чтобы выделить функции, задуманные как конструкторы, их называют с большой буквы: Animal, а не animal.

В результате вызова new Animal("ёжик"); получаем такой объект:

animal = { name: "ёжик", canWalk: true }

Теперь многократными вызовами new Animal с разными параметрами мы можем создать столько объектов, сколько нужно. Поэтому такую функцию и называют конструктором – она предназначена для «конструирования» объектов.

new function() { … }

Иногда функцию-конструктор объявляют и тут же используют, вот так:

```
var animal = new function() {
    this.name = "Васька";
    this.canWalk = true;
};
```

this – это текущий объект при вызове «через точку» и новый объект при конструировании через new.

Метод call

Синтаксис метода call:

func.call(context, arg1, arg2, ...)

При этом вызывается функция func, первый аргумент call становится её this, а остальные передаются «как есть».

Вызов func.call(context, a, b...) – то же, что обычный вызов func(a, b...), но с явно указанным this(=context).

При помощи call можно легко взять метод одного объекта, в том числе встроенного, и вызвать в контексте другого. Это называется «одалживание метода» (на англ. method borrowing).

Метод apply

Если нам неизвестно, с каким количеством аргументов понадобится вызвать функцию, можно использовать более мощный метод: apply. Вызов функции при помощи func.apply работает аналогично func.call, но принимает массив аргументов вместо списка.

Пример потери контекста

Этот код выведет «Привет» через 1000 мс, то есть 1 секунду:

```
setTimeout(function() {
    alert( "Привет" );
}, 1000);
```

Попробуем сделать то же самое с методом объекта, следующий код должен выводить имя пользователя через 1 секунду:

```
var user = {
    firstName: "Вася",
    sayHi: function() {
        alert( this.firstName );
    }
};

setTimeout(user.sayHi, 1000);   // undefined (не Вася!)
```

При запуске кода выше через секунду выводится вовсе не "Вася", а undefined! Это произошло потому, что в примере выше setTimeout получил функцию user.sayHi, но не её контекст.

Самый простой вариант решения – это обернуть вызов в анонимную функцию:

```
var user = {
    firstName: "Вася",
    sayHi: function() {
        alert( this.firstName );
    }
};

setTimeout(function() {
    user.sayHi();  // Вася
}, 1000);
```

Теперь код работает, так как user достаётся из замыкания.

- Функция сама по себе не запоминает контекст выполнения.
- Чтобы гарантировать правильный контекст для вызова obj.func(), нужно использовать функцию-обёртку, задать её через анонимную функцию:

  setTimeout(function() { obj.func(); })
- Либо использовать bind:

  setTimeout(obj.func.bind(obj));
- Вызов bind часто используют для привязки функции к контексту, чтобы затем присвоить её в обычную переменную и вызывать уже без явного указания объекта.
- Вызов bind также позволяет фиксировать первые аргументы функции («каррировать» её), и таким образом из общей функции получить её «частные» варианты – чтобы использовать их многократно без повтора одних и тех же аргументов каждый раз.

Декоратор – это обёртка над функцией, которая модифицирует её поведение. При этом основную работу по-прежнему выполняет функция.

Оператор **typeof** надежно работает с примитивными типами, кроме null, а также с функциями. Он возвращает для них тип в виде строки. Но все объекты, включая массивы и даты для typeof – на одно лицо, они имеют один тип 'object'.

Оператор **instanceof** позволяет проверить, создан ли объект данной функцией, причём работает для любых функций – как встроенных, так и наших.

```
function User() {}

var user = new User();

alert( user instanceof User );    // true
```

Для написания полиморфных (это удобно!) функций нам нужна проверка типов.

- Для примитивов с ней отлично справляется оператор typeof.

  У него две особенности:

  Он считает null объектом, это внутренняя ошибка в языке.
  Для функций он возвращает function, по стандарту функция не считается базовым типом, но на практике это удобно и полезно.
- Для встроенных объектов мы можем получить тип из скрытого свойства [[Class]], при помощи вызова {}.toString.call(obj).slice(8, -1). Для конструкторов, которые объявлены нами, [[Class]] всегда равно "Object".
- Оператор obj instanceof Func проверяет, создан ли объект obj функцией Func, работает для любых конструкторов.
- И, наконец, зачастую достаточно проверить не сам тип, а просто наличие нужных свойств или методов. Это называется «утиная типизация».

JSON (JavaScript Object Notation) используется для представления объектов в виде строки. Это один из наиболее удобных форматов данных при взаимодействии с JavaScript. Если нужно с сервера взять объект с данными и передать его клиенту, то в качестве промежуточного формата – для передачи по сети, почти всегда используют именно его.

Почти все языки программирования имеют библиотеки для преобразования объектов в формат JSON. Основные методы для работы с JSON в JavaScript – это:

- JSON.parse – читает объекты из строки в формате JSON.
- JSON.stringify – превращает объекты в строку в формате JSON, используется, когда нужно из JavaScript передать данные по сети.

Вызов JSON.parse(str) превратит строку с данными в формате JSON в JavaScript-объект/массив/значение. Например:

```
var numbers = "[0, 1, 2, 3]";

numbers = JSON.parse(numbers);

alert( numbers[1] );    // 1
```

Или так:

```
var user = '{ "name": "Вася", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';

user = JSON.parse(user);

alert( user.friends[1] );   // 1
```

Данные могут быть сколь угодно сложными, объекты и массивы могут включать в себя другие объекты и массивы. Главное, чтобы они соответствовали формату.

JSON-объекты ≠ JavaScript-объекты

Объекты в формате JSON похожи на обычные JavaScript-объекты, но отличаются от них более строгими требованиями к строкам – они должны быть именно в двойных кавычках.

```
{
  name: "Вася",   // ошибка: ключ name без кавычек!
  "surname": 'Петров',  // ошибка: одинарные кавычки у значения 'Петров'!
  "age": 35,   // .. а тут всё в порядке
  "isAdmin": false   // и тут тоже всё ок
}
```

Кроме того, в формате JSON не поддерживаются комментарии. Он предназначен только для передачи данных.

Есть нестандартное расширение формата JSON, которое называется JSON5 (http://json5.org/) и как раз разрешает ключи без кавычек, комментарии и т.п, как в обычном JavaScript. На данном этапе это отдельная библиотека.

Сериализация, метод JSON.stringify

Метод JSON.stringify(value, replacer, space) преобразует («сериализует») значение в JSON-строку.

```
var event = {
    title: "Конференция",
    date: "сегодня"
};

var str = JSON.stringify(event);

alert( str );    // {"title":"Конференция","date":"сегодня"}

// Обратное преобразование

event = JSON.parse(str);
```

При сериализации объекта вызывается его метод toJSON.

Методы JSON.parse и JSON.stringify позволяют интеллектуально преобразовать объект в строку и обратно.

setTimeout

Следующий код вызовет func() через одну секунду:

```
function func() {
      alert( 'Привет' );
}

setTimeout(func, 1000);
```

С передачей аргументов (не сработает в IE9-):

```
function func(phrase, who) {
       alert( phrase + ', ' + who );
}

setTimeout(func, 1000, "Привет", "Вася");   // Привет, Вася
```

Если первый аргумент является строкой, то интерпретатор создаёт анонимную функцию из этой строки. То есть такая запись тоже сработает:

```
setTimeout("alert('Привет')", 1000);
```

Функция setTimeout возвращает числовой идентификатор таймера timerId, который можно использовать для отмены действия.

В следующем примере мы ставим таймаут, а затем удаляем (передумали). В результате ничего не происходит.

```
var timerId = setTimeout(function() { alert(1) }, 1000);

alert(timerId); // число - идентификатор таймера

clearTimeout(timerId);

alert(timerId); // всё ещё число, оно не обнуляется после отмены
```

setInterval

Метод setInterval имеет синтаксис, аналогичный setTimeout.

Смысл аргументов – тот же самый. Но, в отличие от setTimeout, он запускает выполнение функции не один раз, а регулярно повторяет её через указанный интервал времени. Остановить исполнение можно вызовом clearInterval(timerId).

Следующий пример при запуске станет выводить сообщение каждые две секунды, пока не пройдёт 5 секунд:

```
// начать повторы с интервалом 2 сек

var timerId = setInterval(function() {
    alert( "тик" );
}, 2000);

// через 5 сек остановить повторы

setTimeout(function() {
    clearInterval(timerId);
    alert( 'стоп' );
}, 5000);
```

Функция eval(code) позволяет выполнить код, переданный ей в виде строки. Этот код будет выполнен в текущей области видимости.

```
var a = 1;

(function() {
    var a = 2;
    eval(' alert(a) ');    // 2
})();
```

Но он может не только выполнить код, но и вернуть результат. Вызов eval возвращает последнее вычисленное выражение, например:

```
alert( eval('1+1') ); // 2
```

eval применяется очень редко. Есть даже такое выражение «eval is evil» (eval – зло). Причина проста: когда-то JavaScript был гораздо более слабым языком, чем сейчас, и некоторые вещи без eval было сделать невозможно. Но те времена давно прошли. И теперь найти тот случай, когда действительно надо выполнить код из строки – это надо постараться.

- Функция eval(str) выполняет код и возвращает последнее вычисленное выражение. В современном JavaScript она используется редко.
- Вызов eval может читать и менять локальные переменные. Это – зло, которого нужно избегать.
- Для выполнения скрипта в глобальной области используются трюк с window.eval/execScript. При этом локальные переменные не будут затронуты, так что такое выполнение безопасно и иногда, в редких архитектурах, может быть полезным.
- Если выполняемый код всё же должен взаимодействовать с локальными переменными – используйте new Function. Создавайте функцию из строки и передавайте переменные ей, это надёжно и безопасно.

Перехват ошибок, "try..catch"

Как бы мы хорошо ни программировали, в коде бывают ошибки. Или, как их иначе называют, «исключительные ситуации» (исключения). Обычно скрипт при ошибке, как говорят, «падает», с выводом ошибки в консоль. Но бывают случаи, когда нам хотелось бы как-то контролировать ситуацию, чтобы скрипт не просто «упал», а сделал что-то разумное. Для этого в JavaScript есть замечательная конструкция try..catch.

Конструкция try..catch состоит из двух основных блоков: try, и затем catch:

```
try {
    // код ...
} catch (err) {
   // обработка ошибки
}
```

Работает она так:

1. Выполняется код внутри блока try.
2. Если в нём ошибок нет, то блок catch(err) игнорируется, то есть выполнение доходит до конца try и потом прыгает через catch.
3. Если в нём возникнет ошибка, то выполнение try на ней прерывается, и управление прыгает в начало блока catch(err).

   При этом переменная err (можно выбрать и другое название) будет содержать объект ошибки с подробной информацией о произошедшем.

Таким образом, при ошибке в try скрипт не «падает», и мы получаем возможность обработать ошибку внутри catch.

- Пример без ошибок: при запуске сработают alert (1) и (2):

```
try {
    alert('Начало блока try'); // (1) <--
    // .. код без ошибок
    alert('Конец блока try'); // (2) <--
} catch(e) {
    alert('Блок catch не получит управление, так как нет ошибок'); // (3)
}

alert("Потом код продолжит выполнение...");
```

- Пример с ошибкой: при запуске сработают (1) и (3):

```
try {
    alert('Начало блока try'); // (1) <--
    lalala; // ошибка, переменная не определена!
    alert('Конец блока try'); // (2)
} catch(e) {
    alert('Ошибка ' + e.name + ":" + e.message + "\n" + e.stack); // (3) <--
}

alert("Потом код продолжит выполнение...");
```

try..catch подразумевает, что код синтаксически верен

Если грубо нарушена структура кода, например не закрыта фигурная скобка или где-то стоит лишняя запятая, то никакой try..catch здесь не поможет. Такие ошибки называются синтаксическими, интерпретатор не может понять такой код. Здесь же мы рассматриваем ошибки семантические, то есть происходящие в корректном коде, в процессе выполнения.

try..catch работает только в синхронном коде

Ошибку, которая произойдёт в коде, запланированном «на будущее», например в setTimeout, try..catch не поймает.

В примере выше мы видим объект ошибки. У него есть три основных свойства:

- name
  Тип ошибки. Например, при обращении к несуществующей переменной: "ReferenceError".
- message
  Текстовое сообщение о деталях ошибки.
- stack
  Везде, кроме IE8-, есть также свойство stack, которое содержит строку с информацией о последовательности вызовов, которая привела к ошибке.

Обработка ошибок

LoftSchool
от мыслителя к создателю

Содержание

1

Виды ошибок

Синтаксические ошибки - возникают из-за неверного конструирования программного кода. Причиной их возникновения могут быть опечатки, возникающие при наборе текста, ошибки в идентификаторах, пропущенные знаки препинания и несоответствие скобок.

```
function func({

}
```

Логические ошибки

Приложение или функция выполняются не так, как было задумано. Например, при суммировании числа и строки

```
var number = '2';

for (var i = 1; i < 5; i++) {
   console.log(i + number); //12
                            //22
                            //32
}
```

Ошибки выполнения

Ошибки выполнения (run-time error) - синтаксический корректный оператор пытается выполнить некорректное действие. Пример - недопустимые вызовы функции, несоответствие типов данных , деление на ноль, присвоение не объявленной переменной

```
var b;
c = b + 1; //переменная c не была объявлена
```

2

Контекст выполнения

Контекст выполнения функции — это одно из фундаментальных понятий в JavaScript. Контекстом еще часто называют значение переменной this внутри функции. Также необходимо отметить что понятие «контекст выполнения» и «область видимости» — это не одно и то же.

Область видимости

Область видимости или ***(scope)*** - определяет доступ к переменным при вызове функции и является уникальной для каждого вызова.

Каждое выполнение функции хранит все переменные в специальном объекте с кодовым именем (scope), который нельзя получить в явном виде, но он есть.

Каждый вызов var - всего лишь создает новое свойство этого объекта, а любое упоминание переменной - первым делом ищется в свойствах этого объекта.

Все изменения локальных переменных являются изменениями свойств этого неявного объекта.

Обычно после того, как функция закончила выполнение, ее область видимости (scope) т.е весь набор локальных переменных убивается.

this

this — это ссылка на объект, который «вызывает» код в данный момент. Значение this чаще всего определяется тем, как вызывается функция. Когда функция вызывается как метод объекта, переменная **this** приобретает значение ссылки на объект, который вызывает этот метод:

```
var user = {
   name: 'John Smith',
   getName: function() {
       console.log(this.name);
   }
};

user.getName(); // John Smith
```

Когда мы вызываем функцию как функцию (не как метод объекта), эта функция будет выполнена в глобальном контексте. Значением переменной this в данном случае будет ссылка на глобальный объект. Однако, если функция вызывается как функция в строгом режиме (strict mode) — значением **this** будет **undefined**.

Контекст выполнения содержит и область видимости, и аргументы функции, и переменную this.

Код в JavaScript может быть одного из следующих типов:

eval-код	код, выполняющийся внутри функции eval()
код функции	код, выполняющийся в теле функции
глобальный код	код, не выполняющийся в рамках какой-либо функции

Когда интерпретатор JavaScript выполняет код, по умолчанию контекстом выполнения является глобальный контекст. Каждый вызов функции приводит к созданию нового контекста выполнения

```
//глобальный контекст выполнения
var hello = 'Hello';

var user = function() { //контекст выполнения функции
   var name = 'John Smith';

   var getName = function() {//контекст выполнения функции
       return name;
   }
}
```

Каждый раз, когда создается новый контекст выполнения, он добавляется в верхнюю часть стека выполнения. Браузер всегда будет выполнять код в текущем контексте выполнения, который находится на вершине стека выполнения. После завершения, контекст будет удален из верхней части стека и управление вернется к контексту выполнения ниже.

Главные моменты:

- Однопоточность — JavaScript работает в однопоточном режиме, т.е. только одна операция может быть выполнена в определенный момент времени.
- Синхронное выполнение кода — код выполняется синхронно, т.е. следующая операция не выполняется до завершения предыдущей.
- Один глобальный контекст выполнения.
- Бесконечное количество контекстов выполнения функции. Каждый вызов функции создает новый контекст выполнения, даже если функция рекурсивно вызывает сама себя.

В интерпретаторе JavaScript каждое создание контекста выполнения происходит в два этапа: этап создания (когда функция только вызвана, но код внутри нее еще не выполняется) и этап выполнения. На этапе создания интерпретатор сначала создает объект переменных (также называемый объект активации), который состоит из всех переменных, объявлений функций и аргументов, определенных внутри контекста выполнения. Затем инициализируется область видимости, и в последнюю очередь определяется значение переменной this. На этапе выполнения внутренним переменным присваивается значение, код интерпретируется и выполняется.

3

Объект Error

Объект Error создается при возникновении ошибки в процессе выполнения сценария и содержит информацию об ошибке, которая используется операторами обработки исключений. Конструктор Error создаёт объект ошибки.

Объект Error также может использоваться в качестве базового для пользовательских исключений.

```
new Error([message[, fileName[, lineNumber]]])
```

Параметры

message - Необязательный параметр. Человеко-читаемое описание ошибки.

fileName - Необязательный параметр. Значением по умолчанию является имя файла, содержащего код, вызвавший конструктор Error().

lineNumber - Необязательный параметр. Значением по умолчанию является номер строки, содержащей вызов конструктора Error().

Стандартные свойства

message - описание ошибки.
Это свойство содержит краткое описание ошибки. Как правило, это основной источник информации о произошедшей ошибке.

```
var e = new Error("Произошла проблема");

console.log(e.message); //произошла проблема
```

name - название типа ошибки.
По умолчанию, объекты класса Error получают значение "**Error**". Однако, можно его поменять на другое:

```
var e = new Error("Malformed input")

console.log(e.name); //выведет Error

e.name = "ParseError"
```

Обычно, **объект Error** создается с намерением возбудить ошибку с помощью ключевого слова ***throw***.

Обработка ошибки производится с помощью конструкции **try...catch:**

4

throw, try, catch

Ключевое слово throw используется для выбрасывания исключения. Ловлей занимается кусок кода, обёрнутый в **блок try**, за которым следует **catch**. Когда код в блоке **try** выкидывает исключение, выполняется блок **catch**. Переменная, указанная в скобках, будет привязана к значению исключения. После завершения выполнения блока **catch**, или же если блок **try** выполняется без проблем, выполнение переходит к коду, лежащему после инструкции ***try/catch***.

```
try {
   throw new Error('Ошибка!');
} catch (e) {
   console.log(e.name + ': ' + e.message); //выведет
Error: Ошибка!
}
```

Пример: обработка определённой ошибки

Также возможно обрабатывать только какой-то определённый вид ошибок, с помощью ключевого слова **instanceof**:

```
try {
   foo.bar();
} catch (e) {
   if (e instanceof EvalError) {
       console.log(e.name + ': ' + e.message);
   } else if (e instanceof RangeError) {
       console.log(e.name + ': ' + e.message);
   }
   // ... и т.д.
}
```

У инструкции try есть ещё одна особенность. За ней может следовать блок **finally**, либо вместо **catch**, либо вместе с **catch**. Блок **finally** означает "*выполнить код в любом случае после выполнения блока try*". Если функции надо что-то подчистить, то подчищающий код нужно включать в блок **finally**.

```
try {
   //какой-то код с ошибкой
} catch (e) {
   console.log(e.message);
}
finally {
   console.log('код выполнен');
}
```

5

Консоль в Chrome Developer Tools

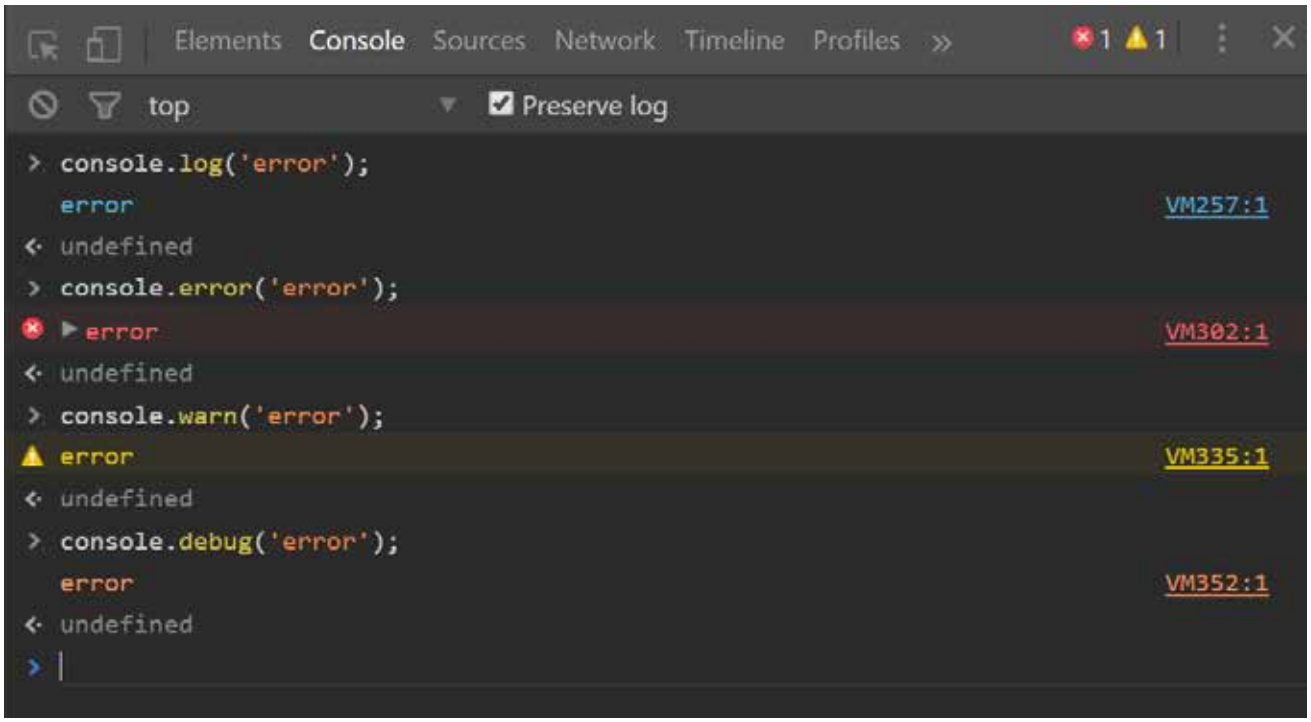
Консоль в Chrome Developer Tools - интерактивный JavaScript-интерпретатор. Консоль дает разработчику доступ к ряду удобных и полезных функций для логирования и отображения ошибок

- console.log()
- console.error()
- console.warn()
- console.debug()

Базовые функции вывода в консоль, позволяют выводить в консоль произвольные сообщения.

Отличаются классификацией выводимых сообщений:

`error()`	помечает сообщения как ошибки
`warn()`	помечает сообщения как предупреждения
`debug()`	помечает сообщения как отладочные



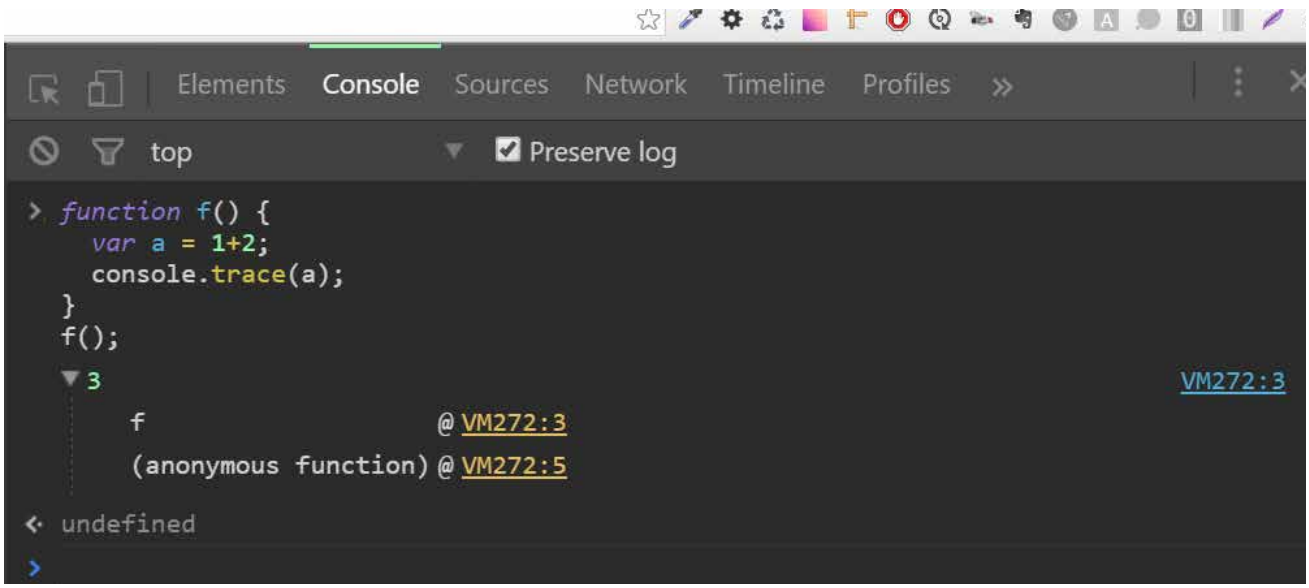
Маски вывода

`%s`	выводит значение как строку
`%d, %i`	выводит значение как целое число
`%f`	выводит значение как число с плавающей запятой
`%o`	выводит значение как элемент DOM
`%0`	выводит значение как объект JavaScript
`%c`	применяет к значению заданные CSS стили

```
console.log('%0', document.body);
```

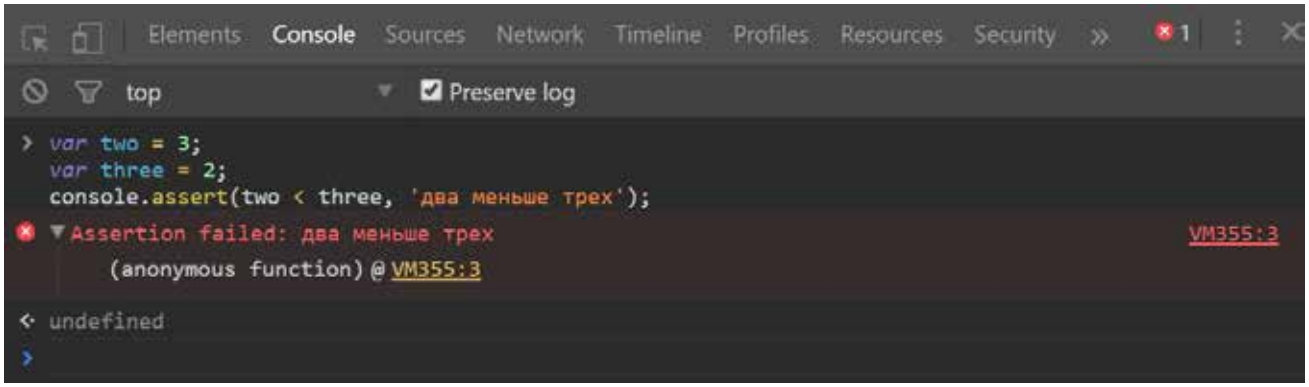
console.trace()

Выводит стек вызовов из точки в коде, где был вызван метод. Стек вызовов включает имена файлов и номера строк плюс счетчик вызовов метода trace() из одной и той-же точки.



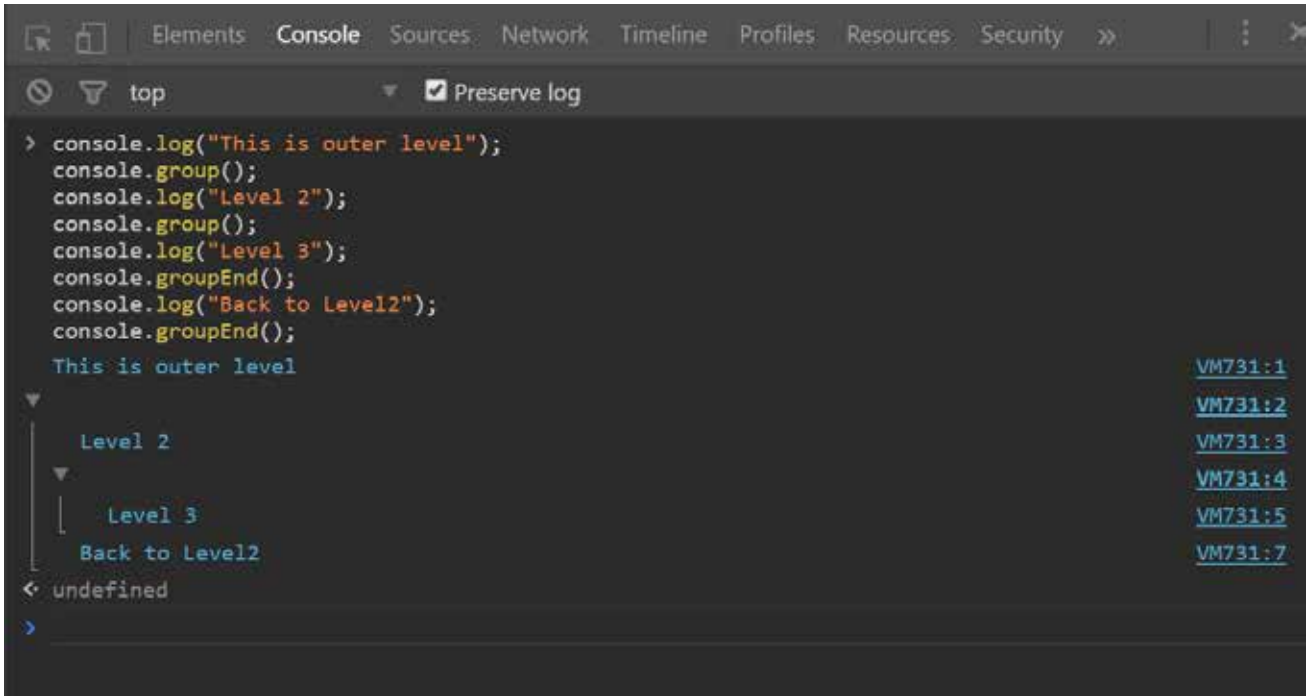
console.assert()

Функция assert проверяет выражение, переданное первым параметром, и если выражение ложно, записывает в консоль ошибку вместе со стеком вызовов:



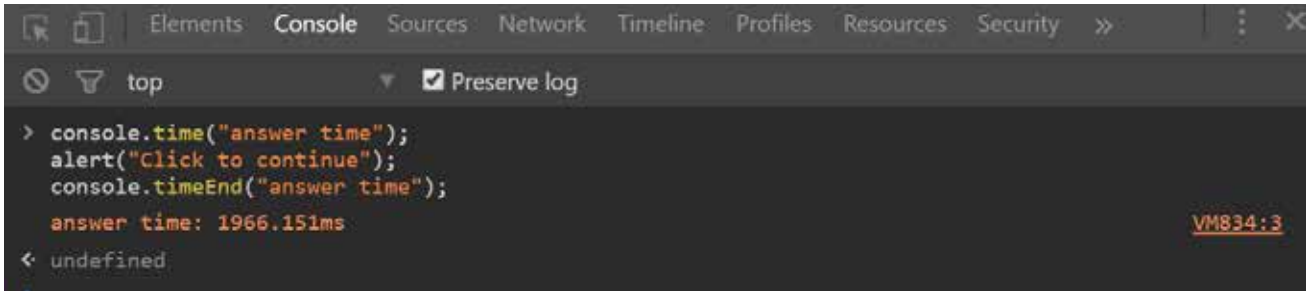
console.group() и console.groupEnd();

Функции для группировки вывода. Функция **group()** открывает группу сообщений, в качестве параметра принимает название группы (поддерживается форматирование, как в **console.log()**. **groupEnd()** закрывает группу



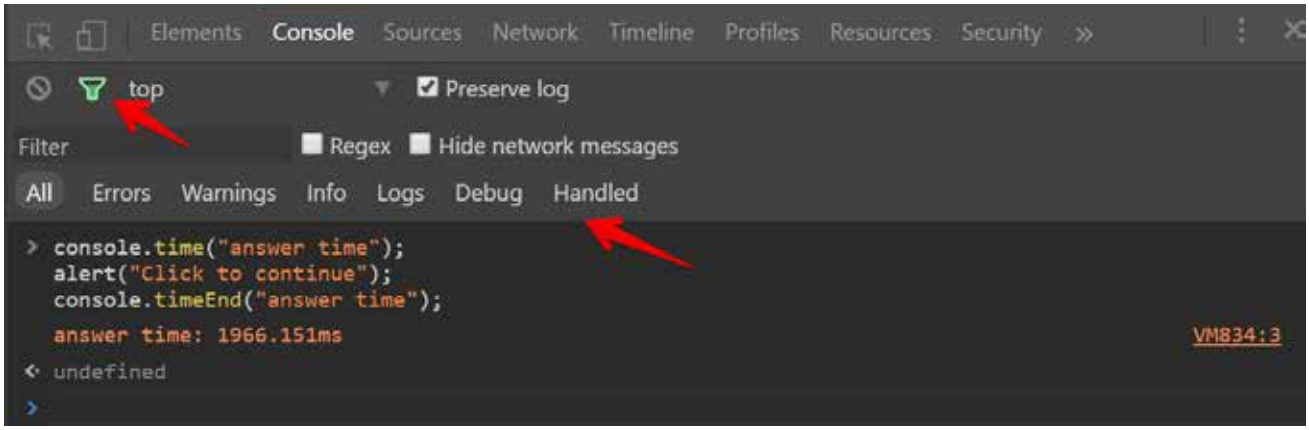
console.time() и console.timeEnd()

Функции для вычисления времени исполнения кода. Функция time() запускает таймер, а функция timeEnd() останавливает таймер и выводит его значение. Обе функции принимают название таймера в качестве обязательного параметра.



Фильтр сообщений

На вкладке консоли расположен фильтр сообщений по типу.



All	соответствует всем сообщениям
Errors	ошибкам и выводу функции console.error()
Warnings	предупреждениям и выводу функции console.warn()
Logs	выводу функции console.log()
Debug	выводу функций console.debug(), console.timeEnd() и прочей информации

$(), $$() и $x()

Функции, упрощающие выборку элементов, работают только в консоли. ***Функция $()*** возвращает первый элемент, соответствующий переданному селектору. Вторым параметром можно передать контекст поиска:
Функция **$$()** аналогична **$()**, но возвращает все найденные элементы

Функция **$x()** возвращает все элементы, соответствующие выражению **XPath**. Вторым параметром можно передать контекст:

$0 — $4

Консоль хранит в памяти ссылки на последние пять элементов, выделенных во вкладке Элементов (Elements). Для доступа к ним используются переменные **$0, $1, $2, $3 и $4**. **$0** хранит ссылку на текущий выделенный элемент, **$1** — на предыдущий и так далее.

$_

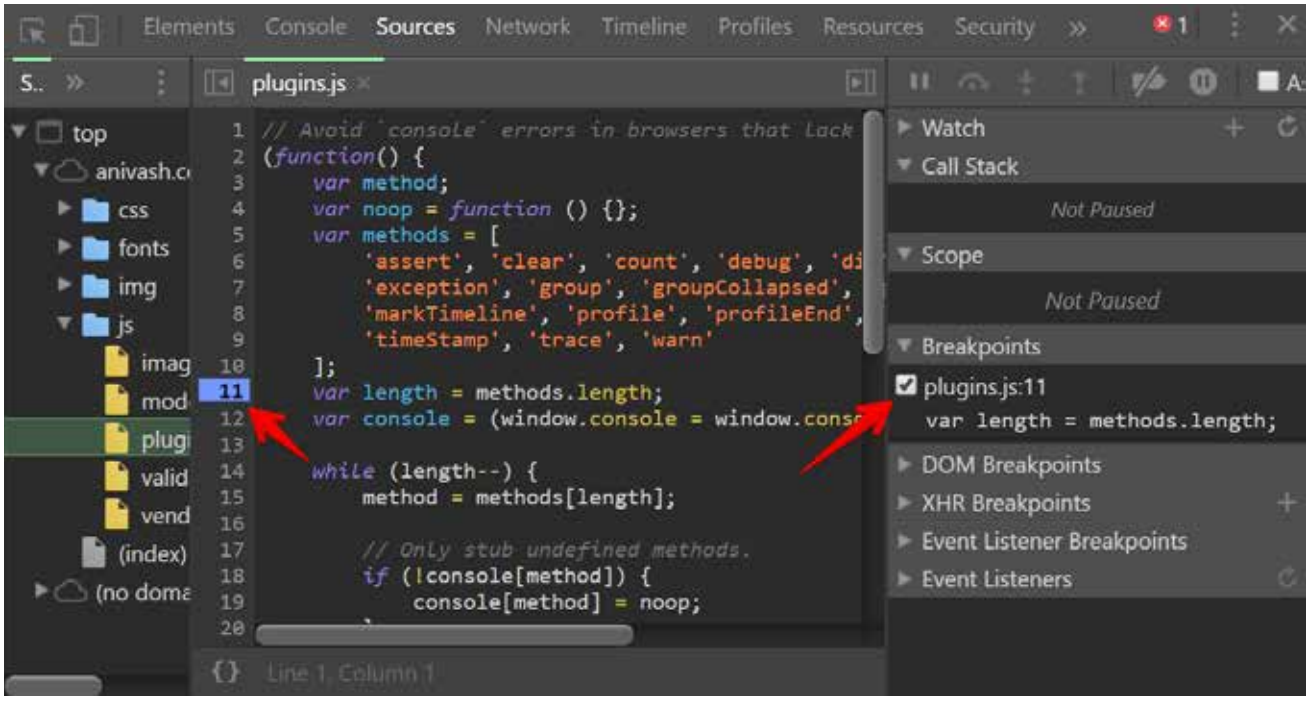
Переменная **$_** хранит результат отработки последней команды в консоли. Это позволяет использовать результат выполнения одной команды в другой команде. Попробуйте выполнить эти команды по очереди:

```
$('body');
$_; //выведет описание элемента body
```

6

Точки остановки (Breakpoints)

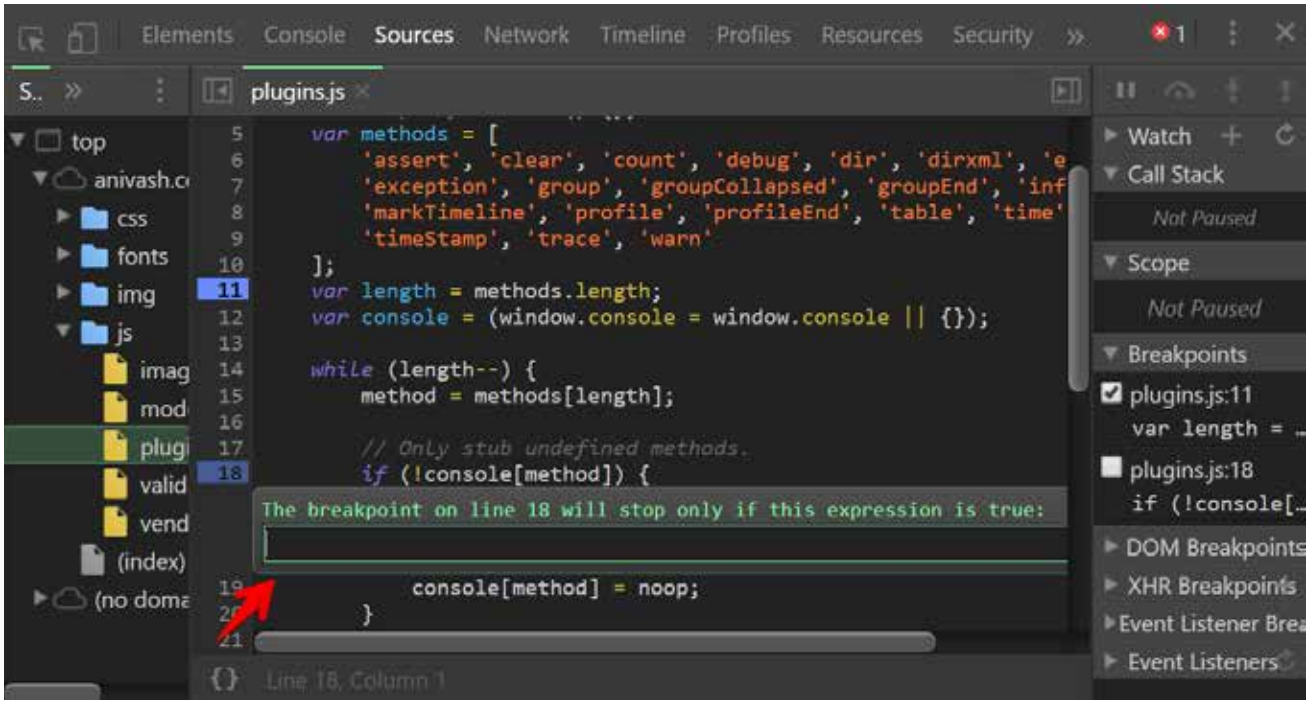
Самый простой вариант остановки кода — задать обычную точку останов-ки. Точка остановки привязывается к строке. Как только интерпретатор JavaScript достигает этой строки, код встает на паузу. В режиме паузы можно посмотреть значения всех переменных, как локальных, так и глобальных, а также исполнять код пошагово.



Также в режиме паузы работает консоль, более того, в консоли доступен контекст функции, в которой остановлен код. Это очень удобно, ведь можно, не отключая паузу, отладить код, избегая зацикливания на внесении измене-ний, сохранении и перезагрузке страницы.

Условные точки остановки (Conditional breakpoints)

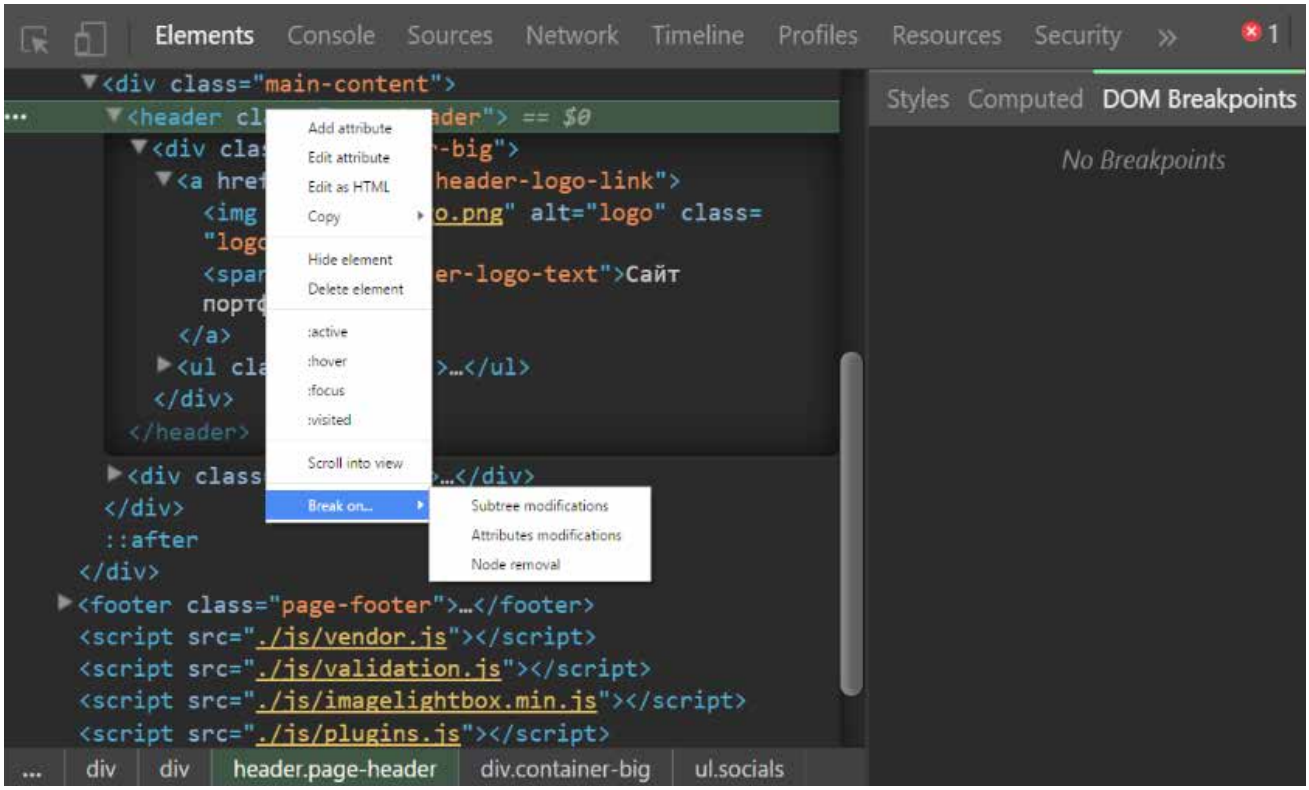
Точки остановки, несомненно, очень полезный инструмент, но часто строка кода исполняется тысячи раз, а с точки зрения отладки интересна лишь при определенных условиях. В такой ситуации на помощь приходят условные точки остановки.



Чтобы создать условную точку остановки, нужно кликнуть правой кнопкой мышки по номеру строки, выбрать «Add conditional breakpoint...» и ввести выражение. Если в момент исполнения кода выражение истинно, исполнение будет приостановлено.

Точки остановки DOM (DOM Breakpoints)

Часто бывают ситуации, когда какой-то скрипт модифицирует элемент на странице или его содержимое, но идентифицировать обидчика не получается. Для таких случаев в Chrome Developer Tools предусмотрены точки остановки DOM. Они позволяют приостановить исполнение кода в случае изменения атрибутов элемента («Attributes modifications»), изменений в дереве дочерних элементов («Subtree modifications») либо удаления элемента («Node removal»).

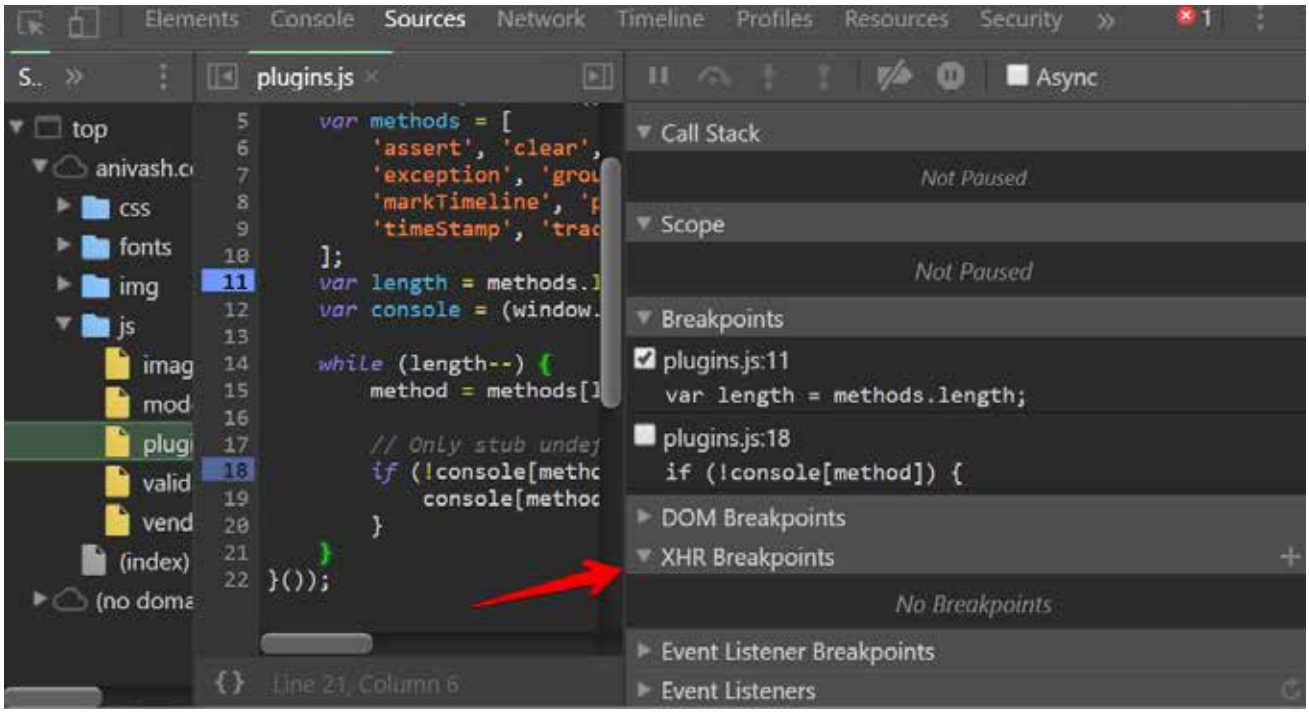


Для создания точки остановки DOM необходимо найти элемент во вкладке **«Elements»**, щелкнуть его правой кнопкой мыши и в открывшемся меню выбрать пункт **«Break on...»**.

Точки остановки XHR (XHR Breakpoints)

При разработке веб-приложений регулярно возникает необходимость отлаживать ajax-запросы. Задача усложняется, если ошибка возникает лишь при обращении к определенному URL-адресу. На помощь в этой ситуации приходят точки остановки XHR (XmlHttpRequest). Они останавливают исполнение кода в момент отправки ajax-запроса, позволяя задать URL-адрес либо его часть.

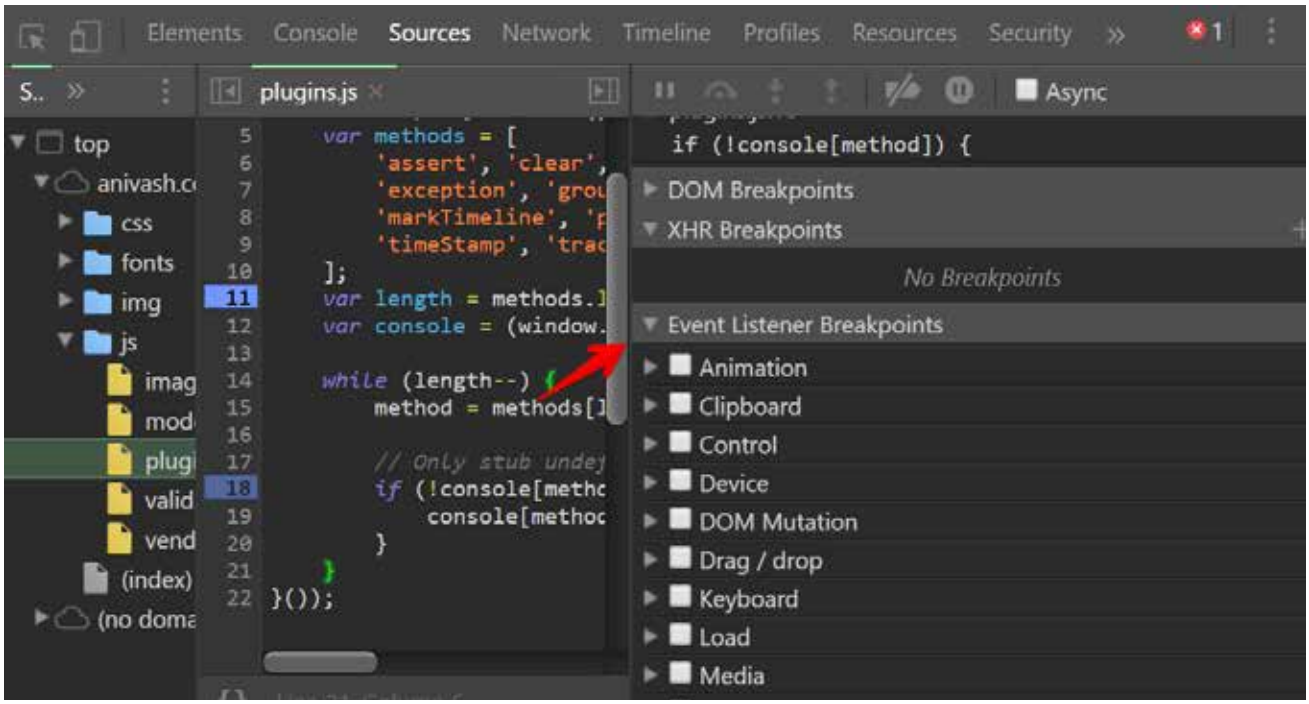
Установить точку остановки XHR можно со вкладки «Sources», нажав на иконку плюсика напротив заголовка «XHR Breakpoints» в правой панели.



Точки остановки по событиям (Event Listener Breakpoints)

Допустим, один из скриптов где-то создает обработчик событий, и необходимо его найти, зная лишь название события. Для этого идеально подойдет точка остановки по событиям. Chrome Developer Tools позволяет установить точку остановки как на конкретное событие (например click или keypress), так и на целую группу событий (например «Mouse» или «Keyboard»).

Чтобы установить точку остановки по событиям, нужно перейти во вкладку «Sources» и в правой панели под заголовком «Event Listener Breakpoints» выбрать интересующие события.



Оператор debugger;

Когда браузер достигает строчки **debugger;** в любом коде, он автоматически останавливает выполнение скрипта в этой точке и переходит на вкладку Скриптов (Sources).