**Лекція 1.Вступ у JavaScript**

1. [**Що**](https://learn.javascript.ru/intro#%D1%87%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-javascript) **таке JavaScript.**
2. **Найпростіший код на JavaScript. Зовнішні скрипти.**
3. **Змінні та типи даних в JavaScript**

**1.** [**Що**](https://learn.javascript.ru/intro#%D1%87%D1%82%D0%BE-%D1%82%D0%B0%D0%BA%D0%BE%D0%B5-javascript) **таке JavaScript.**

JavaScript спочатку створювалась для того, щоб зробити web-сторінки «живими». Програми на цій мові називаються скриптами. У браузері вони підключаються безпосередньо до HTML і, як тільки завантажується сторінка - тут же виконуються.

Програми на JavaScript - звичайний текст. Вони не вимагають якоїсь спеціальної підготовки.

В цьому плані JavaScript сильно відрізняється від іншої мови, яка називається Java.

***Чому JavaScript?***

Коли створювали мову JavaScript, у ній спочатку була інша назва: «LiveScript». Але тоді була дуже популярна мова Java, і маркетологи вирішили, що схожа назву зробить нову мову більш популярною.

Планувалося, що JavaScript буде такою собі «молодшою сестрою» Java. Однак, історія розпорядилася по-своєму, JavaScript сильно виросла, і зараз це абсолютно незалежна мова, зі своєю специфікацією, яка називається ECMAScript, і до Java не має ніякого відношення.

JavaScript може виконуватися не тільки в браузері, а де завгодно, потрібна лише спеціальна програма - інтерпретатор. Процес виконання скрипта називають «інтерпретацією».

Сучасні інтерпретатори перед виконанням перетворять JavaScript в машинний код або близько до нього, оптимізують, а вже потім виконують. І навіть під час виконання намагаються оптимізувати. Тому JavaScript працює дуже швидко.

У всі основні браузери вбудований інтерпретатор JavaScript, саме тому вони можуть виконувати скрипти на сторінці. Але, зрозуміло, JavaScript можна використовувати не тільки в браузері. Це повноцінна мова, програми на якій можна запускати і на сервері, і навіть в пральній машинці, якщо в ній встановлений відповідний інтерпретатор.

***Що вміє JavaScript?***

Сучасна JavaScript - це «безпечна» мова програмування загального призначення. Вона не надає низькорівневих засобів роботи з пам'яттю, процесором, так як спочатку була орієнтована на браузери, в яких це не потрібно.

Що ж стосується інших можливостей - вони залежать від оточення, в якому запущена JavaScript. У браузері JavaScript вміє робити все, що відноситься до маніпуляції зі сторінкою, взаємодії з відвідувачем і, в якійсь мірі, з сервером:

• Створювати нові HTML-теги, видаляти існуючі, змінювати стилі елементів, ховати, показувати елементи і т.п.

• Реагувати на дії відвідувача, обробляти кліки миші, переміщення курсора, натискання на клавіатуру і т.п.

• Посилати запити на сервер і завантажувати дані без перезавантаження сторінки (ця технологія називається "AJAX").

• Отримувати і встановлювати cookie, запитувати дані, виводити повідомлення і багато багато іншого!

***Що НЕ вміє JavaScript?***

JavaScript - швидка і потужна мова, але браузер накладає на її виконання деякі обмеження. Це зроблено для безпеки користувачів, щоб зловмисник не міг за допомогою JavaScript отримати особисті дані або якось нашкодити комп'ютеру користувача.

Цих обмежень немає там, де JavaScript використовується поза браузером, наприклад на сервері. Крім того, сучасні браузери надають свої механізми по установці плагінів і розширень, які володіють розширеними можливостями, але вимагають спеціальних дій по установці від користувача.

Більшість можливостей JavaScript в браузері обмежена поточним вікном і сторінкою.

 • JavaScript не може читати / записувати довільні файли на жорсткий диск, копіювати їх або викликати програми. Вона не має прямого доступу до операційної системи.

Сучасні браузери можуть працювати з файлами, але ця можливість обмежена спеціально виділеної Директорією - «пісочницею». Можливості щодо доступу до пристроїв також опрацьовуються в сучасних стандартах і частково доступні в деяких браузерах.

• JavaScript, що працює в одній вкладці, не може спілкуватися з іншими вкладками і вікнами, за винятком випадку, коли вона сама відкрила це вікно або декілька вкладок з одного джерела (однаковий домен, порт, протокол).

Є способи це обійти, але вони вимагають спеціального коду на обидва документи, які знаходяться в різних вкладках або вікнах. Без нього, з міркувань безпеки, залізти з однієї вкладки в іншу за допомогою JavaScript не можна.

• З JavaScript можна легко посилати запити на сервер, з якого прийшла сторінка. Запит на інший домен теж можливий, але менш зручний, оскільки тут є обмеження безпеки.

***У чому унікальність JavaScript?***

Є як мінімум три чудових особливості JavaScript:

• Повна інтеграція з HTML / CSS.

• Прості речі робляться просто.

•Підтримується всіма поширеними браузерами і включена за замовчуванням.

Цих трьох речей одночасно немає більше ні в одній браузерної технології. Тому JavaScript і є найпоширенішим засобом створення браузерних інтерфейсів.

Багато можливості HTML5 все ще в розробці, але браузери поступово починають їх підтримувати. JavaScript стає все більш і більш потужна і можливості браузера ростуть в сторону десктопних додатків. JavaScript стає все швидша і стабільніша, в мову додаються нові можливості.

Дуже важливо те, що нові стандарти HTML5 та ECMAScript зберігають максимальну сумісність з попередніми версіями. Це дозволяє уникнути неприємностей з уже існуючими додатками.

***Плагіни і додаткові налаштування для браузера***

Всі сучасні браузери надають можливість написати плагіни. Для цього можна використовувати як JavaScript (Chrome, Opera, Firefox), так і мову С (ActiveX для Internet Explorer). Ці плагіни можуть як відображати вміст спеціального формату (плагін для програвання музики, для показу PDF), так і взаємодіяти зі сторінкою. Як і в ситуації з Java-аплетом, у них широкі можливості, але відвідувач поставить їх у тому випадку, якщо вам довіряє.

Мова JavaScript унікальна завдяки своїй повній інтеграції з HTML / CSS. Вона працює майже у всіх відвідувачів. Але хороший JavaScript-програміст не повинен забувати і про інші технології.

Адже наша мета - створення хороших додатків, і тут Flash, Java і браузерні розширення мають свої унікальні можливості, які можна використовувати разом з JavaScript.

Що ж стосується CoffeeScript, TypeScript та інших мов, побудованих над JavaScript - вони можуть бути дуже корисні. Рекомендується подивитися їх, хоча б у загальних рисах, але, звичайно, вже після освоєння самого JavaScript.

1. **Найпростіший код на JavaScript. Зовнішні скрипти.**

Програми на мові JavaScript можна вставити в будь-яке місце HTML за допомогою тегу SCRIPT. Наприклад:

<!DOCTYPE HTML>

<html>

<head>

<!-- Тег meta для вказування кодування -->

<meta charset="utf-8">

</head>

<body>

<p>Початок документу...</p>

<script>

alert( 'Привіт, світ!' );

</script>

<p>...Кінець документу</p>

</body>

</html>

Цей приклад використовує такі елементи:

*<script> ... </ script>*

Тег *script* містить виконуваний код. Попередні стандарти HTML вимагали обов'язкового зазначення атрибута type, але зараз він вже не потрібен. Досить просто <script>.

Браузер, коли бачить <script>:

1. Починає відображати сторінку, показує частину документа до script

2. Зустрівши тег script, перемикається в JavaScript-режим і не показує, а виконує його вміст.

3. Після завершення виконання, повертається назад в HTML-режим і тільки тоді відображає частину документа.

*alert (повідомлення)*

Показує вікно з повідомленням і чекає, поки відвідувач не натисне «Ок».

***Кодування і тег meta***

При спробі зробити такий же файл у себе на диску і запустити, ви можете зіткнутися з проблемою - виводяться «кракозябли», «квадратики» і «вопросики» замість українського тексту.

Щоб все було добре, потрібно:

1. Переконатися, що в HEAD є рядок <meta charset = "utf-8">. Якщо ви будете відкривати файл з диска, то саме він вкаже браузеру кодування.

2. Переконатися, що редактор зберіг файл саме в кодуванні UTF-8, а не, скажімо, в windows-1251.

Вказівка ​​кодування - частина звичайного HTML.

***Сучасна розмітка для SCRIPT***

У старих скриптах оформлення тега SCRIPT було трохи складніше. У застарілих інструкціях можна зустріти наступні елементи:

Атрибут *<script type = ...>*

На відміну від HTML5, стандарт HTML 4 вимагав обов'язкового зазначення цього атрибута. Виглядав він так: type = "text / javascript". Якщо вказати інше значення type, то скрипт виконаний не буде.

У сучасній розробці атрибут type необов'язковий.

Атрибут *<script language = ...>*

Цей атрибут призначений для вказівки мови, на якому написаний скрипт. За замовчуванням, мова - JavaScript, так що і цей атрибут ставити необов'язково.

Отже, для вставки скрипта ми просто пишемо *<script>,* без додаткових атрибутів і коментарів.

***Зовнішні скрипти, порядок виконання***

Якщо JavaScript-коду багато - його виносять в окремий файл, який підключається в HTML:

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

Тут /path/to/script.js - це абсолютний шлях до файлу, який містить скрипт (з кореня сайту).

Браузер сам завантажить скрипт і виконає.

Можна вказати і повний URL, наприклад:

<script src="https://cdnjs.cloudflare.com/ajax/libs/lodash.js/ 4.3.0/lodash.js"> </script>

Ви також можете використовувати шлях щодо поточної сторінки. Наприклад, src = "lodash.js" позначає файл з поточної директорії.

Щоб підключити декілька скриптів, використовуйте кілька ключових слів:

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

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

...

*На замітку:*

Як правило, в HTML пишуть тільки найпростіші скрипти, а складні виносять в окремий файл.

Браузер завантажить його тільки перший раз і в подальшому, при правильному налаштуванні сервера, буде брати зі свого кеша.

Завдяки цьому один і той же великий скрипт, що містить, наприклад, бібліотеку функцій, може використовуватися на різних сторінках без повного перезавантаження з сервера.

Якщо зазначений атрибут src, то вміст тега ігнорується.

В одному тезі SCRIPT не можна одночасно підключити зовнішній скрипт і вказати код.

Ось так не спрацює:

script src="file.js">

alert(1); // оскільки вказаний src, то внутрішня частина тегу ігнорується

</script>

Потрібно вибрати: або SCRIPT йде з src, або містить код. Тег вище слід розбити на два: один - з src, інший - з кодом, ось так:

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

<script>

alert( 1 );

</script>

**Асинхронні скрипти: *defer / async***

Браузер завантажує і відображає HTML поступово. Особливо це помітно при повільному інтернет-з'єднанні: браузер не чекає, поки сторінка завантажиться повністю, а показує ту частину, яку встиг завантажити.

Якщо браузер бачить тег <script>, то він за стандартом зобов'язаний спочатку виконати його, а потім показати решту сторінки.

Наприклад, в прикладі нижче - поки усі кролики не будуть пораховані - нижній <p> не буде показаний:

<!DOCTYPE HTML>

<html>

<head>

<meta charset="utf-8">

</head>

<body>

<p>Починаємо рахувати:</p>

<script>

alert( 'Перший кролик!' );

alert( 'Другий кролик!' );

alert( 'Третій кролик!' );

</script>

<p>Кролики підраховані!</p>

</body>

</html>

Таку поведінку називають «синхронною». Як правило, вона цілком нормальна, але є важливий наслідок.

Якщо скрипт - зовнішній, то поки браузер не виконає його, він не покаже частину сторінки під ним.

Тобто, в такому документі, поки не завантажиться і не виконається big.js, вміст <body> буде приховано

<html>

<head>

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

</head>

<body>

Цей текст не буде показаний, поки браузер не виконає big.js.

</body>

</html>

І тут питання - чи дійсно ми цього хочемо? Тобто, чи дійсно решту сторінки не можна показувати до завантаження скрипта?

Є ситуації, коли ми не тільки НЕ хочемо такої затримки, але вона навіть небезпечна.

Наприклад, якщо ми підключаємо зовнішній скрипт, який показує рекламу або вставляє лічильник відвідувань, а потім йде наша сторінка. Звичайно, неправильно, що поки лічильник або реклама не довантажити - решта сторінки не показується. Лічильник відвідувань не повинен ніяк затримувати відображення сторінки сайту. Реклама теж не повинна гальмувати сайт і порушувати його функціональність.

А що, якщо сервер, з якого завантажується зовнішній скрипт, перевантажений? Відвідувач в цьому випадку може чекати дуже довго!

Ось приклад, з подібним скриптом (варто штучна затримка завантаження):

<p> Важлива інформація буде непереливки, поки не завантажиться скрипт.</p>

<script src="https://js.cx/hello/ads.js?speed=0"></script>

<p>... Важлива інформація!</p>

Що робити?

Можна поставити всі подібні скрипти в кінець сторінки - це зменшить проблему, але не позбавить від неї повністю, якщо скриптів кілька. Припустимо, в кінці сторінки 3 скрипта, і перший з них гальмує - виходить, інші два його чекатимуть - теж недобре.

Крім того, браузер дійде до скриптів, розташованих в кінці сторінки, вони почнуть завантажуватися тільки тоді, коли вся сторінка завантажиться. А це не завжди правильно. Наприклад, лічильник відвідуваності найбільш точно спрацює, якщо завантажити його раніше.

Тому «розташувати скрипти внизу» - не кращий вихід.

Кардинально вирішити цю проблему допоможуть атрибути *async* або *defer*:

*Атрибут* ***async***

Підтримується всіма браузерами, крім IE9-. Скрипт виконується повністю асинхронно. Тобто, при виявленні <script async src = "..."> браузер не зупиняє обробку сторінки, а спокійно працює далі. Коли скрипт буде завантажений - він виконається.

*Атрибут* ***defer***

Підтримується всіма браузерами, включно з більшістю старих IE. Скрипт також виконується асинхронно, не примушує чекати сторінку, але є дві відмінності від *async*.

Перше - браузер гарантує, що відносний порядок скриптів з *defer* буде збережений.

Тобто, в такому коді (з *async*) першим спрацює той скрипт, який раніше завантажиться:

<script src="1.js" async></script>

<script src="2.js" async></script>

А в такому коді (з *defer*) першим спрацює завжди 1.js, а скрипт 2.js, навіть якщо завантажився раніше, буде його чекати.

<script src="1.js" defer></script>

<script src="2.js" defer></script>

Тому атрибут *defer* використовують в тих випадках, коли другий скрипт 2.js залежить від першого 1.js, наприклад - використовує щось, описане першим скриптом.

Друга відмінність - скрипт з *defer* спрацює, коли весь HTML-документ буде оброблений браузером.

Наприклад, якщо документ досить великий ...

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

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

Багато багато багато літер

...

то скрипт async.js виконається, як тільки завантажиться - можливо, до того, як весь документ готовий. А defer.js почекає готовності всього документа.

Це буває зручно, коли ми в скрипті хочемо працювати з документом, і повинні бути впевнені, що він цілком отриманий.

При одночасному вказівці *async* і *defer* в сучасних браузерах буде використаний лише async, в IE9- - тільки *defer* (не розуміє *async*).

**Атрибути async / defer - тільки для зовнішніх скриптів**

Атрибути async / defer працюють тільки в тому випадку, якщо призначені на зовнішні скрипти, тобто мають *src*.

При спробі призначити їх на звичайні скрипти <script> ... </ script>, вони будуть проігноровані.

Той же приклад з *async*:

  <p> Важлива інформація тепер не чекає, поки завантажиться скрипт...</p>

<script async src="https://js.cx/hello/ads.js?speed=0"></script>

<p>...Важлива інформація!</p>

При запуску ви побачите, що вся сторінка відобразилася тут же, а *alert* із зовнішнього скрипта з'явиться пізніше, коли завантажиться скрипт.

Більшість сучасних систем реклами та лічильників знають про ці атрибути і використовують їх.

Перед вставкою зовнішнього тега <script> розуміє програміст завжди перевірить, чи є у нього подібний атрибут. Інакше повільний скрипт може затримати завантаження сторінки.

Для просунутого читача, який знає, що теги <script> можна додавати на сторінку в будь-який момент за допомогою самого javascript, зауважимо, що скрипти, додані таким чином, ведуть себе так само, як async. Тобто, виконуються як тільки буде завантажено, без збереження відносного порядку.

Якщо ж потрібно зберегти порядок виконання, тобто додати кілька скриптів, які виконаються строго один за іншим, то використовується властивість script.async = false.

Виглядає це приблизно так:

function addScript(src){

var script = document.createElement('script');

script.src = src;

script.async = false; // щоб гарантувати порядок

document.head.appendChild(script);

}

addScript('1.js'); // // завантажуватися ці скрипти почнуть відразу addScript('2.js'); // // завантажуватися ці скрипти почнуть відразу

addScript('3.js'); // але, гарантовано, в порядку 1 -> 2 -> 3

***Висновок***

• Скрипти вставляються на сторінку як текст в тезі <script>, або як зовнішній файл через <script src = "шлях"> </ script>

• Спеціальні атрибути async і defer використовуються для того, щоб поки вантажиться зовнішній скрипт - браузер показав іншу (наступну за ним) частину сторінки. Без них цього не відбувається.

• Різниця між *async* і *defer*: атрибут *defer* зберігає відносну послідовність скриптів, а *async* - немає. Крім того, *defer* завжди чекає, поки весь HTML-документ буде готовий, а  - немає.

1. **Змінні та типи даних в JavaScript**

Залежно від того, для чого ви робите скрипт, прийдеться працювати з інформацією. Якщо це електронний магазин - то це товари, кошик. Якщо чат - відвідувачі, повідомлення і так далі. Щоб зберігати інформацію, використовуються змінні.

Змінна складається з імені та виділеної області пам'яті, яка йому відповідає. Для оголошення або, іншими словами, створення змінної використовується ключове слово *var*:

var message;

Після оголошення, можна записати в змінну дані:

var message;

message = 'Hello'; // збережемо в змінну рядок

Ці дані будуть збережені у відповідній області пам'яті і надалі доступні при зверненні до імені:

var message;

message = 'Hello!';

alert( message ); // виведе вміст змінної

Для стислості можна поєднати оголошення змінної і запис даних:

var message = 'Hello!';

Можна навіть оголосити кілька змінних відразу:

var user = 'John', age = 25, message = 'Hello';

Найпростіше зрозуміти змінну, якщо уявити її як «коробку» для даних, з унікальним ім'ям.

Наприклад, змінна message - це коробка, в якій зберігається значення "Hello!":

В коробку можна покласти будь-яке значення, а пізніше - поміняти його. Значення в змінної можна змінювати скільки завгодно разів:

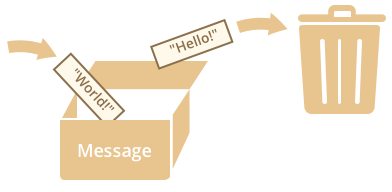
var message;

message = 'Hello!';

message = 'World!'; // замінили значення

alert( message );

При зміні значення старе вміст змінної видаляється.



Можна оголосити дві змінні і копіювати дані з однієї в іншу:

var hello = 'Hello world!';

var message;

// скопіювали значення

message = hello;

alert( hello ); // Hello world!

alert( message ); // Hello world!

***Імена змінних***

На ім'я змінної в JavaScript накладено всього два обмеження.

1. Ім'я може складатися з: букв, цифр, символів *$* і \_

2. Перший символ не повинен бути цифрою.

Наприклад:

var myName;

var test123;

Що особливо цікаво - долар '$' і знак підкреслення '\_' є такими ж звичайними символами, як букви:

var $ = 1; // оголосили змінну з ім’ям '$'

var \_ = 2; // змінна з ім’ям '\_'

alert( $ + \_ ); // 3

А такі змінні були б неправильними:

var 1a; // початок не може бути цифрою

var my-name; // дефіс '-' не є дозволеним символом

Регістр букв має значення. Змінні apple і AppLE - дві різні змінні. Українські літери допустимі, але не рекомендуються

***Зарезервовані імена***

Існує список зарезервованих слів, які не можна використовувати для змінних, так як вони використовуються самим мовою, наприклад: *var, class, return, export* і ін.

Наприклад, такий приклад видасть синтаксичну помилку:

var return = 5; // помилка

alert(return);

***Константи***

Константа - це змінна, яка ніколи не змінюється. Як правило, їх називають великими буквами, через підкреслення. наприклад:

var COLOR\_RED = "#F00";

var COLOR\_GREEN = "#0F0";

var COLOR\_BLUE = "#00F";

var COLOR\_ORANGE = "#FF7F00";

var color = COLOR\_ORANGE;

alert( color ); // #FF7F00

Технічно, константа є звичайною змінною, тобто її можна змінити. Але ми домовляємося цього не робити.

Навіщо потрібні константи? Чому б просто не писати var color = "# FF7F00"?

1. По-перше, константа COLOR\_ORANGE - це зрозуміле ім'я. За присвоєння var color = "# FF7F00" незрозуміло, що колір - помаранчевий. Іншими словами, константа COLOR\_ORANGE є «зрозумілим псевдонімом» для значення # FF7F00.

2. По-друге, помилка в рядку, особливо такої складної як # FF7F00, може бути не помічена, а в імені константи її допустити куди складніше.

Константи використовують замість рядків і чисел, щоб зробити програму більш зрозумілою і уникнути помилок.

**Висновок**

• В JavaScript можна оголошувати змінні для зберігання даних. Це робиться за допомогою var.

• Технічно, можна просто записати значення і без оголошення змінної, проте з ряду причин це не рекомендується.

• Разом з оголошенням можна відразу присвоїти значення: var x = 10.

• Змінні, які названі ВЕЛИКИМИ ЛІТЕРАМИ, є константами, тобто ніколи не змінюються. Як правило, вони використовуються для зручності, щоб було менше помилок.

***Шість типів даних, typeof***

В JavaScript існує кілька основних типів даних.

***Число «number»***

var n = 123;

n = 12.345;

Єдиний тип число використовується як для цілих, так і для дрібних чисел.

Існують спеціальні числові значення Infinity (нескінченність) і NaN (помилка обчислень).

Наприклад, нескінченність Infinity виходить при розподілі на нуль:

alert( 1 / 0 ); // Infinity

Помилка обчислень NaN буде результатом некоректної математичної операції, наприклад:

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

Ці значення формально належать типу «число», хоча, звичайно, числами в їх звичайному розумінні не є.

Особливості роботи з числами в JavaScript розібрані в главі Числа.

***Рядок «string»***

var str = "Мама мила раму";

str = 'Такі лапки теж підходять';

В JavaScript одинарні та подвійні лапки рівноправні. Можна використовувати або ті або інші.

Тип *символ* не існує, є тільки *рядок*.

У деяких мовах програмування є спеціальний тип даних для одного символу. Наприклад, в мові С це char. В JavaScript є тільки тип «рядок» string. Що, треба сказати, цілком зручно.

***Булевий (логічний) тип «boolean»***

У нього всього два значення: *true* (істина) і *false* (брехня).

Як правило, такий тип використовується для зберігання значення типу так / ні, наприклад:

var checked = true; // // поле форми позначено галочкою

checked = false; // поле форми не містить галочки

***Спеціальне значення «null»***

Значення null не відноситься ні до одного з типів вище, а утворює свій окремий тип, що складається з одного значення null:

var age = null;

В JavaScript null не є «посиланням на неіснуючий об'єкт» або «нульовим покажчиком», як в деяких інших мовах. Це просто спеціальне значення, яке має сенс «нічого» або «значення невідомо».

Зокрема, код вище говорить про те, що вік age невідомий.

***Спеціальне значення «undefined»***

Значення undefined, як і null, утворює свій власний тип, що складається з одного цього значення. Воно має сенс «значення не присвоїли».

Якщо змінна оголошена, але в неї нічого не записано, то її значення як раз і є undefined:

var x;

alert( x ); // виведе "undefined"

Можна привласнити undefined і в явному вигляді, хоча це робиться рідко:

var x = 123;

x = undefined;

alert( x ); // "undefined"

В явному вигляді undefined зазвичай не присвоюють, так як це суперечить його змістом. Для запису в змінну «порожнього» або «невідомого» значення використовується null.

***Об'єкти «object»***

Перші 5 типів називають "примітивними".

Окремо стоїть шостий тип: «об'єкти».

Він використовується для колекцій даних і для оголошення більш складних сутностей.

Оголошуються об'єкти за допомогою фігурних дужок {...}, наприклад:

var user = { name: "Вася" };

Ми детально розберемо способи оголошення об'єктів і, взагалі, роботу з об'єктами, пізніше, в розділі Об'єкти як асоціативні масиви.

***Оператор typeof***

Оператор typeof повертає тип аргументу. У нього є два синтаксису: з дужками і без: *typeof x* та *typeof* (*x*). Працюють вони однаково, але перший синтаксис коротше. Результатом typeof є рядок, що містить тип:

typeof undefined // "undefined"

typeof 0 // "number"

typeof true // "boolean"

typeof "foo" // "string"

typeof {} // "object"

typeof null // "object" (1)

typeof function(){} // "function" (2)

Останні два рядки позначені, тому що typeof поводиться в них по-особливому.

1. Результат *typeof null == "object"* - це офіційно визнана помилка в мові, яка зберігається для сумісності. Насправді null - це не об'єкт, а окремий тип даних.

2. Функції ми пройдемо трохи пізніше. Поки лише зауважимо, що функції не є окремим базовим типом в JavaScript, а підвидом об'єктів. Але typeof виділяє функції окремо, повертаючи для них "*function*". На практиці це досить зручно, оскільки дозволяє легко визначити функцію. Отже є 5 «примітивних» типів: number, string, boolean, null, undefined і 6-й тип - об'єкти object.

Оператор typeof x дозволяє з'ясувати, який тип знаходиться в x, повертаючи його у вигляді рядка