1. couchdb  
2. angularjs  
3. ООП, Наследование  
4. Область видимости, Контекст, call, apply  
5. Замыкания, самовызывающиеся функции, Инкапсуляция  
6. Основы и особенности javascript. Типы данных. 'use strict'. Всплытия (hoisting). Работа с DOM.  
7. Работа с асинхронностью, промисы, стек вызова функций, stack overflow  
8. Design Patterns. Примеры на js  
9. ES6  
10. События, обработка событий, Event Bubbling, Event Capturing.  
11. Ajax, Sockets

1. couchdb <http://anton.shevchuk.name/php/couchdb-for-developers/> <https://ru.wikipedia.org/wiki/CouchDB>  
2. Angularjs <http://habrahabr.ru/post/149060/>  
3. ООП, Наследование <http://javascript.ru/tutorial/object/inheritance>  
4. Область видимости, Контекст, call, apply <http://habrahabr.ru/post/239863/>  
5. Замыкания, самовызывающиеся функции, Инкапсуляция <http://habrahabr.ru/post/239863/>  
6. Основы и особенности javascript. Типы данных. 'use strict'. Всплытия (hoisting). Работа с DOM.

<https://learn.javascript.ru/javascript-specials> <http://habrahabr.ru/post/127482/> <http://javascript.ru/tutorial/dom/intro>   
7. Работа с асинхронностью, промисы, стек вызова функций, stack overflow

<https://learn.javascript.ru/promise> <http://javascript.ru/unsorted/async>  
8. Design Patterns. Примеры на js <http://habrahabr.ru/post/132472/>  
9. ES6 <http://habrahabr.ru/post/241275/>  
10. События, обработка событий, Event Bubbling, Event Capturing. <http://javascript.ru/tutorial/events/intro>  
11. Ajax, Sockets <https://ru.wikipedia.org/wiki/WebSocket> <https://ru.wikipedia.org/wiki/AJAX>

1.Couchdb

CouchDB

**CouchDB** — [документо-ориентированная](https://ru.wikipedia.org/wiki/%D0%94%D0%BE%D0%BA%D1%83%D0%BC%D0%B5%D0%BD%D1%82%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94) [система управления базами данных](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D1%83%D0%BF%D1%80%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F_%D0%B1%D0%B0%D0%B7%D0%B0%D0%BC%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85) с [открытым исходным кодом](https://ru.wikipedia.org/wiki/%D0%9E%D1%82%D0%BA%D1%80%D1%8B%D1%82%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), не требующая [описания схемы данных](https://ru.wikipedia.org/wiki/%D0%A1%D1%85%D0%B5%D0%BC%D0%B0_%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85), распространяется [свободно](https://ru.wikipedia.org/wiki/%D0%A1%D0%B2%D0%BE%D0%B1%D0%BE%D0%B4%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), написана на языке [Erlang](https://ru.wikipedia.org/wiki/Erlang). Впервые вышла в 2005 году, с 2008 года — проект [фонда Apache](https://ru.wikipedia.org/wiki/Apache_Software_Foundation).

Подход[[править](https://ru.wikipedia.org/w/index.php?title=CouchDB&veaction=edit&vesection=1) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=CouchDB&action=edit&section=1)]

Реализована в рамках подхода [NoSQL](https://ru.wikipedia.org/wiki/NoSQL). Для хранения данных используется [JSON](https://ru.wikipedia.org/wiki/JSON), для реализации [MapReduce](https://ru.wikipedia.org/wiki/MapReduce)-запросов — [JavaScript](https://ru.wikipedia.org/wiki/JavaScript). Одной из особенностей СУБД является поддержка [репликации](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BF%D0%BB%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0)) с несколькими ведущими узлами.

CouchDB можно рассматривать как сервер веб-приложений; для реализации этой идеи в CouchDB встроен производительный [веб-сервер](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80), а программный код, как и данные, сохраняется в той же базе данных. Для автоматизации работы с приложениями используется утилита CouchApp.

Следуя подходу NoSQL, CouchDB не хранит данные и связи в таблицах. Вместо этого каждая база данных — набор независимых документов. Каждый документ содержит свои собственные данные и независимую схему. Приложение может получить доступ к нескольким базам данных, например, хранящейся на мобильном телефоне пользователя и на сервере. Метаданные документа содержат информацию о версии, позволяя объединять данные и разрешать любые противоречия, которые могли появиться в момент, когда базы данных были разъединены.

Для управления конкурентным доступом используется механизм [MVCC](https://ru.wikipedia.org/wiki/MVCC), благодаря чему возможно избежать необходимости блокировки файла базы данных во время записи. Разрешение конфликтов относится к сфере ответственности логики приложения, разрешение конфликта обычно включает в себя объединение данных в один документ, а затем старый документ удаляется.

Архитектура системы[[править](https://ru.wikipedia.org/w/index.php?title=CouchDB&veaction=edit&vesection=2) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=CouchDB&action=edit&section=2)]

Подобно иным документно-ориентированным СУБД ([Mnesia](https://ru.wikipedia.org/wiki/Mnesia), [Lotus Notes](https://ru.wikipedia.org/wiki/Lotus_Notes), [MongoDB](https://ru.wikipedia.org/wiki/MongoDB)) и в отличие от [реляционных СУБД](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BB%D1%8F%D1%86%D0%B8%D0%BE%D0%BD%D0%BD%D0%B0%D1%8F_%D0%A1%D0%A3%D0%91%D0%94), CouchDB предназначена для работы с полуструктурированной информацией и имеет следующие особенности:

* данные сохраняются не в строках и колонках, а в виде [JSON](https://ru.wikipedia.org/wiki/JSON)-подобных документов, моделью которых является не таблицы, а [деревья](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2));
* типизация элементов данных, то есть сопоставление отдельным полям документов типов INTEGER, DATE и пр., не поддерживается — вместо этого пользователь может написать функцию-[валидатор](https://ru.wikipedia.org/wiki/%D0%92%D0%B0%D0%BB%D0%B8%D0%B4%D0%B0%D1%82%D0%BE%D1%80_%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B0);
* целостность базы данных обеспечивается исключительно на уровне отдельных записей (но не на уровне связей между ними);
* связи между таблицами или записями принципиально не поддерживаются, соответственно операция объединения (JOIN) между таблицами не определена;
* для построения индексов и выполнения запросов используются функции [представления](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B0%D0%B2%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_(%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) (view)[[4]](https://ru.wikipedia.org/wiki/CouchDB#cite_note-Apache.2C_View_Server-4);
* функции-валидаторы, функции-представления, функции-фильтры сохраняются в текстовом виде в самой базе данных;
* эти функции, как правило, написаны на языках [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) или Erlang, а для их выполнения запускается отдельный сервер запросов, взаимодействие с которым происходит посредством [сокетов](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BA%D0%B5%D1%82_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D1%8B%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81)) и текстового JSON-[протокола](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%82%D0%BE%D0%BA%D0%BE%D0%BB_%D0%BF%D0%B5%D1%80%D0%B5%D0%B4%D0%B0%D1%87%D0%B8_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85);
* каждой базе данных в системе CouchDB соответствует единственное [B-дерево](https://ru.wikipedia.org/wiki/B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE) (не путать с [двоичным деревом](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE));
* каждое B-дерево хранится в виде отдельного файла на диске;
* одновременно может быть запущено несколько потоков для чтения базы данных и только один — для записи;
* целостность базы данных обеспечивается только при записи данных на диск;
* представления хранятся в БД и их [индексы](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81_(%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) обновляются непрерывно, однако при каждом обновлении функций представления или отображения обновляется всё B-дерево целиком;
* при обработке данных с помощью функций-представлений используется упрощённая модель технологии [MapReduce](https://ru.wikipedia.org/wiki/MapReduce), что позволяет производить[параллельные вычисления](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%BB%D0%BB%D0%B5%D0%BB%D1%8C%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F), в том числе и на [многоядерном процессоре](https://ru.wikipedia.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B3%D0%BE%D1%8F%D0%B4%D0%B5%D1%80%D0%BD%D1%8B%D0%B9_%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80);
* [распределение вычислений](https://ru.wikipedia.org/w/index.php?title=%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D0%B5%D0%BD%D0%B8%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9&action=edit&redlink=1) на несколько узлов не поддерживается — вместо этого используется [механизм репликации](https://ru.wikipedia.org/wiki/%D0%A0%D0%B5%D0%BF%D0%BB%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F_(%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%82%D0%B5%D1%85%D0%BD%D0%B8%D0%BA%D0%B0));
* обработка данных с помощью цепочки последовательных функций [MapReduce](https://ru.wikipedia.org/wiki/MapReduce) не поддерживается;
* поддерживается [вертикальное масштабирование](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D1%80%D1%82%D0%B8%D0%BA%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BC%D0%B0%D1%81%D1%88%D1%82%D0%B0%D0%B1%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5);
* внешний интерфейс ([API](https://ru.wikipedia.org/wiki/API)) к данной СУБД построен на основе архитектуры [REST](https://ru.wikipedia.org/wiki/REST), то есть сама база данных, отдельные записи, отображения и запросы — суть ресурсы, которые имеют уникальный адрес ([URL](https://ru.wikipedia.org/wiki/URL)) и поддерживают операции [GET](https://ru.wikipedia.org/wiki/HTTP#GET), [PUT](https://ru.wikipedia.org/wiki/HTTP#PUT), [POST](https://ru.wikipedia.org/wiki/HTTP#POST), [DELETE](https://ru.wikipedia.org/wiki/HTTP#DELETE);
* поэтому для взаимодействия с базой данных было написано много клиентских библиотек, в том числе на таких языках: [JavaScript](https://ru.wikipedia.org/wiki/JavaScript)[[5]](https://ru.wikipedia.org/wiki/CouchDB#cite_note-5), [PHP](https://ru.wikipedia.org/wiki/PHP), [Ruby](https://ru.wikipedia.org/wiki/Ruby), [Python](https://ru.wikipedia.org/wiki/Python) и[Erlang](https://ru.wikipedia.org/wiki/Erlang);
* взаимодействие между отдельными компонентами СУБД, то есть с серверами представлений осуществляется опять-таки с помощью текстового протокола, а данные передаются в формате JSON; это позволило использовать различные языки программирования для написания этих компонентов — [Java](https://ru.wikipedia.org/wiki/Java), [Python](https://ru.wikipedia.org/wiki/Python), [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) и пр.

.

Управление данными[[править](https://ru.wikipedia.org/w/index.php?title=CouchDB&veaction=edit&vesection=5) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=CouchDB&action=edit&section=5)]

CouchDB управляет коллекцией документов [JSON](https://ru.wikipedia.org/wiki/JSON). Документы организуются по представлениям ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *views*). Представления определяются агрегатными функциями и фильтрами, вычисленными параллельно подобно [MapReduce](https://ru.wikipedia.org/wiki/MapReduce).

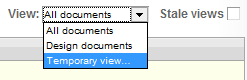
Представления хранятся в базе данных, и их [индексы](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D0%B4%D0%B5%D0%BA%D1%81_(%D0%B1%D0%B0%D0%B7%D1%8B_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D1%85)) обновляются непрерывно. CouchDB поддерживает систему представления, которая использует внешние сокет-серверы и протокол, основанный на JSON. Как следствие, серверы представления были разработаны на множестве языков ([JavaScript](https://ru.wikipedia.org/wiki/JavaScript) — по умолчанию, но есть также [PHP](https://ru.wikipedia.org/wiki/PHP), [Ruby](https://ru.wikipedia.org/wiki/Ruby), [Python](https://ru.wikipedia.org/wiki/Python) и [Erlang](https://ru.wikipedia.org/wiki/Erlang)).

**Доступ по HTTP**[[править](https://ru.wikipedia.org/w/index.php?title=CouchDB&veaction=edit&vesection=6) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=CouchDB&action=edit&section=6)]

Приложения взаимодействуют с CouchDB через HTTP. Ниже представлено несколько примеров с использованием утилиты командной строки cURL. Эти примеры предполагают, что CouchDB работает на localhost (127.0.0.1) на порту 5984.

|  |  |  |
| --- | --- | --- |
| **Действие** | **Запрос** | **Ответ** |
| Доступ к информации о сервере | curl http://127.0.0.1:5984/ | {  "couchdb": "Welcome",  "version":"1.1.0"  } |
| Создание базы данных под именем **wiki** | curl -X PUT http://127.0.0.1:5984/wiki | {"ok": **true**} |
| Попытка создать вторую базу данных под названием **wiki** | curl -X PUT http://127.0.0.1:5984/wiki | {  "error":"file\_exists",  "reason":"The database could not be created, the file already exists."  } |
| Получаем информацию о базе данных **wiki** | curl http://127.0.0.1:5984/wiki | {  "db\_name": "wiki",  "doc\_count": 0,  "doc\_del\_count": 0,  "update\_seq": 0,  "purge\_seq": 0,  "compact\_running": **false**,  "disk\_size": 79,  "instance\_start\_time": "1272453873691070",  "disk\_format\_version": 5  } |
| Удаление базы данных **wiki** | curl -X DELETE http://127.0.0.1:5984/wiki | {"ok": **true**} |
| Создаем документ, прося CouchDB, чтобы предоставил id документа | curl -X POST -H "Content-Type: application/json" --data **\**  '{ "text" : "Wikipedia on CouchDB", "rating": 5 }' **\**  http://127.0.0.1:5984/wiki | {  "ok": **true**,  "id": "123BAC",  "rev": "946B7D1C"  } |

### View

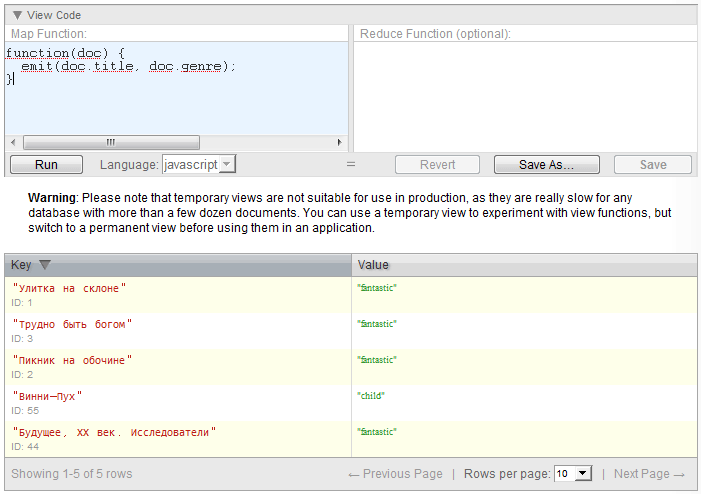
Но вернемся к Fucon’у — настал черед создать view, идем в пункт “Temporary View…”:  


Затем создаем map функцию:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 1  2  3 | function(doc) {    emit(doc.title, doc.genre);  } |

Функция **emit** принимает в качестве параметров ключ и значение, данный параметры могут быть простыми, как в примере выше, или составными — массив или объект. Историю почему функция носит такое имя я рассказать не могу, зваться бы ей push



Cохраняем view как документ \_design/books с именем genre — таки view это обычный документ в нашей БД, можем его даже пощупать:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | {     "\_id": "\_design/books",     "\_rev": "1-532d13f2b2ef9dea495df8230f31b0bf",     "language": "javascript",     "views": {         "genre": {             "map": "function(doc) {\n  emit(doc.title, doc.genre);\n}"         }     }  } |

Результатом нашей работы будет следующий набор данных:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10 | // URL: <http://localhost:5984/simple/_design/books/_view/genre>  // UTF я привел к русскому    {"total\_rows":5,"offset":0,"rows":[    {"id":"44","key":"Будущее, ХХ век. Исследователи","value":"fantastic"},    {"id":"55","key":"Винни-Пух","value":"child"},    {"id":"2","key":"Пикник на обочине","value":"fantastic"},    {"id":"3","key":"Трудно быть богом","value":"fantastic"},    {"id":"1","key":"Улитка на склоне","value":"fantastic"}  ]} |

### Тем кто с SQL знаком

#### **ORDER BY**

CouchDB сортирует выборку по ключу передаваемому в функцию emit первым параметром. Следовательно, если нам надо отсортировать по названию книги, тоmap функцию придется изменить:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 1  2  3 | function(doc) {    emit(doc.title, {title:doc.title, isbn:doc.ISBN});  } |

Так же можно задать составной ключ:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 1  2  3 | function(doc) {    emit([doc.genre, doc.title], {title:doc.title, isbn:doc.ISBN});  } |

Для обратного порядка есть параметр **?descending=true**

#### **WHERE**

Большинство логики ложится непосредственно на функцию map — именно она в ответе за условия поиска:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 1  2  3  4 | function(doc) {      if (doc.authors.length > 1)          emit(doc.\_id, {title:doc.title, isbn:doc.ISBN});  } |

Но так же есть возможность простой фильтрации по ключу:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 1  2  3  4 | function(doc) {      // ключом будет цена книги      emit(doc.price, {title:doc.title, isbn:doc.ISBN});  } |

Получить книги с определенной ценой: **?key=235**  
Получить все книги из диапазона цен: **?startkey=200&endkey=300**

Если у нас составной ключ, то startkey и endkey должны быть составными: **?startkey=[100]&endkey=[300]** (фильтр накладывается лишь на первый элемент составного ключа)

Если мы имеем дело с текстовым ключом, то можно получить конструкцию аналогичную LIKE “Ul%” используя следующий запрос: **?startkey=”ul”&endkey=”UL\ufff0″**.

Порядок сортировки ключей следующий:

` ^ \_ - , ; : ! ? . ' " ( ) [ ] { } @ \* / \ & # % + < = > | ~ $ 0 1 2 3 4 5 6 7 8 9

a A b B c C d D e E f F g G h H i I j J k K l L m M n N o O p P q Q r R s S t T u U v V w W x X y Y z Z

Больше информации о сортировке найдете в wiki: [View сollation](http://wiki.apache.org/couchdb/View_collation)

#### **LIMIT и OFFSET**

Существуют следующие параметры **?limit=5&skip=10**, вроде то что нужно, но, не ведитесь на эту провокацию, CouchDb все равно будет считывать пропущенные документы, но лишь отображать их не будет. Правильным является использование параметра startkey + limit. Алгоритм следующий:

* Выбираем необходимое количество элементов rows\_per\_page + еще один
* Отображаем rows\_per\_page элементов пользователю
* Запасной элемент (его ключ) используем для получения next\_startkey (т.е. ссылки на следующую страницу)
* Ключ первого элемента используем для получения ссылки на предыдущую страницу

#### **SUM, COUNT**

Для подобных вычислений нам понадобится создать reduce функцию, именно она может обработать результаты работы map функции. Давайте вычислим количество книг:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14 | // map  function(doc) {    // ключом будет жанр книги    emit(doc.\_id, doc.title);  }  // reduce  function(keys,values) {      return values.length;  }  // результат  {"rows":[    {"key":null,"value":4}  ]}  // можно еще упростить данный пример |

Или лучше среднюю стоимость книг:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12 | // map  function(doc) {    emit(doc.\_id, doc.price);  }  // reduce  function(keys,values) {      return Math.floor(sum(values)/values.length);  }  // результат  {"rows":[  {"key":null,"value":233}  ]} |

Если надобности в reduce нет, то можно отключить, используя параметр **?reduce=false**

У функции reduce есть еще третий параметр — rereduce. Насколько я понял, в том случае, когда у нас очень большая база, документы в reduce функцию будут поступать пачками, и пока rereduce=false у нас все хорошо, и функция работает как обычно, но когда rereduce=true то к нам на вход попадут в качестве values промежуточные данные вычислений.

#### **GROUP BY**

Для организации группировки нам понадобится reduce функция и параметр **?group=true**. Приведу пример вычисления средней стоимости книг, с группировкой по жанру:

[?](http://anton.shevchuk.name/php/couchdb-for-developers/)

|  |  |
| --- | --- |
| 01  02  03  04  05  06  07  08  09  10  11  12  13  14  15  16 | // map  function(doc) {    // ключом будет жанр книги    emit(doc.genre, doc.price);  }    // reduce  function(keys,values) {      return Math.floor(sum(values)/values.length);  }    // результат  {"rows":[  {"key":"child","value":145},  {"key":"fantastic","value":263}  ]} |

Так же возможна группировка по составному ключу, тут пригодится параметр **?group\_level=1**, с его помощью можно указывать уровень группировки, приведу пример из руководства:

2. Anagularjs

# AngularJS — фреймворк для динамических веб-приложений от Google

[Программирование](http://habrahabr.ru/hub/programming/)\*, [JavaScript](http://habrahabr.ru/hub/javascript/)\*, [AngularJS](http://habrahabr.ru/hub/angularjs/)\*

AngularJS создан для тех разработчиков, которые считают, что декларативный стиль лучше подходит для создания UI, а императивный — для написания бизнес-логики.

#### **Дзен Angular**

* Хорошо отделять манипуляцию DOM-ом от логики работы приложения. Это существенно улучшает тестируемость кода.
* Хорошо считать, что автоматизированное тестирование приложения настолько же важно, насколько и написание самого приложения. Тестируемость очень сильно зависит от того, как структурирован код.
* Хорошо отделять разработку клиентской части от серверной. Это позволяет вести разработку параллельно и улучшает повторное использование на обеих сторонах.
* Хорошо, когда фреймворк ведет разработчика по всему циклу разработки приложения: от проектирования UI через написание бизнес-логики к тестированию.
* Хорошо, когда распространенные задачи становятся тривиальными, а сложные — упрощаются.

AngularJS представляет собой комплексный фреймворк. В стандартной поставке он предоставляет следующие возможности:

* Все, что вам нужно для создания CRUD-приложений: data-binding, базовые директивы для шаблонов, валидация форм, роутинг, deep linking, повторное использование компонентов, dependency injection, инструменты для взаимодействия с серверными (RESTful) источниками данных.
* Все, что вам нужно для тестирования: средства для модульного тестирование, end-to-end тестирования, mock-и.
* [Шаблон типового приложения](https://github.com/angular/angular-seed), включающего в себя структуру каталогов и тестовые скрипты.

AngularJS разрабатывается сотрудниками Google и используется, как минимум, в одном сервисе Google —[DoubleClick](http://builtwith.angularjs.org/project/DoubleClick).

#### **Примеры**

Простенькая todo-шка

<div ng-app>

<h2>Todo</h2>

<div ng-controller="TodoCtrl">

<span>{{remaining()}} of {{todos.length}} remaining</span>

[ <a href="" ng-click="archive()">archive</a> ]

<ul class="unstyled">

<li ng-repeat="todo in todos">

<input type="checkbox" ng-model="todo.done">

<span class="done-{{todo.done}}">{{todo.text}}</span>

</li>

</ul>

<form ng-submit="addTodo()">

<input type="text" ng-model="todoText" size="30" placeholder="add new todo here">

<input class="btn-primary" type="submit" value="add">

</form>

</div>

</div>

​

function TodoCtrl($scope) {

$scope.todos = [

{text:'learn angular', done:true},

{text:'build an angular app', done:false}];

$scope.addTodo = function() {

$scope.todos.push({text:$scope.todoText, done:false});

$scope.todoText = '';

};

$scope.remaining = function() {

var count = 0;

angular.forEach($scope.todos, function(todo) {

count += todo.done ? 0 : 1;

});

return count;

};

$scope.archive = function() {

var oldTodos = $scope.todos;

$scope.todos = [];

angular.forEach(oldTodos, function(todo) {

if (!todo.done) $scope.todos.push(todo);

});

};

}

В действии можно посмотреть на главной странице [angularjs.org](http://angularjs.org/). Там же представлен еще ряд примеров:

* «The Basics» — простенькая иллюстрация databinding.
* «Add Some Control» — todo-шка, код которой я привел здесь.
* «Wire up a Backend» — простое приложение создания/хранения/редактирования записей с роутингом и хранением данных в [mongolab](https://mongolab.com/).
* «Create Components» — создание повторно используемых компонентов.

Еще примеры:

* todo ([демо](http://todomvc.com/architecture-examples/angularjs/), [код](https://github.com/addyosmani/todomvc/tree/master/architecture-examples/angularjs)) из [todomvc](http://todomvc.com/) — заодно можно посравнивать с другими фреймворками;
* [builtwith.angularjs.org](http://builtwith.angularjs.org/): 18 (на момент написания поста) приложений, для большинства из которых доступен исходный код;
* [AngularUI](https://github.com/angular-ui/angular-ui) — разные фильтры и директивы (во многом UI-шные) от сторонних разработчиков.

#### **Основные понятия AngularJS**

##### **Директивы**

На директивах держится практически вся декларативная часть AngularJS. Именно они используются для обогащения синтаксиса HTML. В процессе компиляции DOM директивы берутся из HTML и исполняются. Директивы могут добавить какое-то новое поведение и/или модифицировать DOM. В стандартную поставку входит достаточно большое количество директив для построения веб приложений. Но ключевой особенностью является возможность разработки своих директив, за счет чего HTML может быть превращен в DSL.  
  
Директивы именуются с помощью lowerCamelCase, например, ngBind. При использовании директиву необходимо именовать в нижнем регистре с использованием в качестве разделителя одного из спец символов::, -, или \_. По желанию для получения валидного кода можно использовать префиксы x- или data-. Примеры:ng:bind, ng-bind, ng\_bind, x-ng-bind и data-ng-bind.  
  
Директивы могут использоваться как элемент (<my-directive></my-directive>), атрибут (<div my-directive="exp"> </div>), в классе (<div class="my-directive: exp;"></div>) или в комментарии (<!-- directive: my-directive exp -->). Это зависит от того, как конкретная директива была разработана.  
  
[Подробнее о директивах в руководстве разработчика](http://docs.angularjs.org/guide/directive).

##### **Scope-ы**

Scope — это объект, имеющий отношение к модели в приложении. Он является контекстом выполнения для выражений. Scope-ы выстраиваются в иерархическую структуру, похожую на DOM. При этом они наследуют свойства от своих родительских scope-ов.  
  
Scope-ы являются как бы «клеем» между контроллером и представлением. В процессе выполнения фазы связывания шаблона директивы устанавливают наблюдение ($watch) за выражениями в рамках scope. $watchдает директивам возможность реагировать на изменения для отображения обновленного значения или каких-либо других действий. И контроллеры, и директивы имеют ссылку на scope, но не имеют ссылок друг на друга. Так контроллеры изолируются от директив и от DOM-а. За счет этого возрастают возможности по тестированию приложения.  
  
[Подробнее о scope-ах в руководстве разработчика](http://docs.angularjs.org/guide/scope).

##### **Сервисы**

Сервисы — singleton-ы, выполняющие какую-либо конкретную задачу, которая является общей для всех или конкретного веб-приложения. Например, [$http сервис](http://docs.angularjs.org/api/ng.$http), который является оберткой над XMLHttpRequest. Несколько примеров других сервисов (полный список смотрите в [документации](http://docs.angularjs.org/api/)):

* [$compile](http://docs.angularjs.org/api/ng.$compile) — компиляция HTML-строки или части DOM-а в шаблон, связывание шаблона с конкретным scope-ом;
* [$cookies](http://docs.angularjs.org/api/ngCookies.$cookies) — предоставляет доступ на чтение/запись к cookies.
* [$location](http://docs.angularjs.org/api/ng.$location) — работа с адресной строкой
* [$resource](http://docs.angularjs.org/api/ngResource.$resource) — фабрика по созданию ресурсных объектов, предназначенных для взаимодействия с серверными (RESTful) источниками данных;

Для использования сервиса необходимо указать его как зависимость для контроллера, другого сервиса, директивы и т.п. AngularJS позаботится обо всем остальном — создании, разрешении зависимостей и т.п.  
  
[Подробнее о сервисах в руководстве разработчика](http://docs.angularjs.org/guide/dev_guide.services.understanding_services).

##### **Фильтры**

Фильтры предназначены для форматирования данных перед отображением их пользователю, а также фильтрации элементов в коллекциях. Примеры фильтров (полный список можно посмотреть в [документации](http://docs.angularjs.org/api/)):[currency](http://docs.angularjs.org/api/ng.filter:currency), [date](http://docs.angularjs.org/api/ng.filter:date), [orderBy](http://docs.angularjs.org/api/ng.filter:orderBy), [uppercase](http://docs.angularjs.org/api/ng.filter:uppercase). Использование фильтров достаточно традиционно: {{ expression | filter1 | filter2 }}  
  
[Подробнее о фильтрах в руководстве разработчика](http://docs.angularjs.org/guide/dev_guide.templates.filters).

##### **Модули**

Приложения в AngularJS не имеют основного исполняемого метода. Вместо этого модуль выполняет роль декларативного описания того, как приложение должно быть загружено. Благодаря этому, например, при написании сценариев тестирования можно подгрузить дополнительные модули, которые переопределят какие-то настройки, облегчая тем самым комплексное (end-to-end) тестирование.  
  
[Подробнее о модулях в руководстве разработчика](http://docs.angularjs.org/guide/module).

##### **Тестирование**

Как пишут разработчики, для тестирования они сделали очень много в AngularJS, поэтому уже ничего не может извинить вас, если вы не будете тестировать свое приложение :)  
  
Пример тестового e2e сценария:

describe('Buzz Client', function() {

it('should filter results', function() {

input('user').enter('jacksparrow');

element(':button').click();

expect(repeater('ul li').count()).toEqual(10);

input('filterText').enter('Bees');

expect(repeater('ul li').count()).toEqual(1);

});

});

Подробнее о [модульном тестировании](http://docs.angularjs.org/guide/dev_guide.unit-testing) и [e2e тестировании](http://docs.angularjs.org/guide/dev_guide.e2e-testing) в руководстве разработчика.

#### **AngularJS Batarang**

Это [расширение для Chrome](https://github.com/angular/angularjs-batarang/blob/master/README.md), которое облегчает отладку AngularJS приложений. Позволяет работать с иерархией scope-ов, дает возможность профилирования приложения, визуализирует зависимости между сервисами, отображает содержимое scope-ов на странице элементов, позволяет выводить и менять значения в scope из консоли. Хорошее текстовое описание на [странице в github](https://github.com/angular/angularjs-batarang/blob/master/README.md). Хорошее видео на [youtube](https://www.youtube.com/watch?v=q-7mhcHXSfM).

3. Наследование в Javascript

Javascript - очень гибкий язык. В отличие от Java, PHP, C++ и многих других языков, где наследование можно делать одним способом - в javascript таких способов много.

На уровне языка реализовано *наследование на прототипах*. С помощью некоторых трюков можно сделать (хотя и не так удобно, как в Java/C++) наследование на классах, объявить приватные свойства объекта и многое другое.

*Корректность этой статьи*

С момента появления эта статья вызвала критику некоторых профессионалов в javascript. Поэтому появилось это небольшое "пред-введение".

Не всё, происходящее при наследовании в javascript, статья описывает абсолютно корректно. В том числе, упрощено описание таких вещей как activation object, scope, prototype встроенных объектов.

Если Вы хотите максимально корректное и точное описание наследования, содержащее все детали - оно дано в [стандарте языка](http://javascript.ru/ecma), параграфы 13.2 и 15.3. Возможно, хороший вариант - прочитать статью для создания общей картины, а потом - уточнить технические детали, прочитав стандарт.

Упрощения в статье не затрагивают сути происходящего и не играют роли при real-life использовании javascript. Если Вы аргументированно считаете, что это не так - напишите комментарий, буду рад принять его к сведению и дополнить статью.

P.S Комментарии типа "А что будет, если прототип функции сделать числом и почему статья этого не описывает, это неправильная статья" - не принимаются. Не делают такого в real-life.

Обязательно пишите в комментах, если что неправильно или непонятно - на все отвечу и поправлю. Автор.

## [Создание объекта. Функция-конструктор](http://javascript.ru/tutorial/object/inheritance#sozdanie-obekta-funkciya-konstruktor)

Любая функция, кроме некоторых встроенных, может создать объект.

Для этого ее нужно вызвать через директиву new.

Например, функция Animal в примере ниже создаст новый объект.

|  |  |  |
| --- | --- | --- |
| 1 | function Animal(name) { | |
| 2 | this.name = name |

|  |  |  |
| --- | --- | --- |
| 3 | this.canWalk = true | |
| 4 | } |

|  |  |
| --- | --- |
| 5 |  |
| 6 | var animal = new Animal("скотинка") | |

Во время работы функции, вызванной директивой new, новосоздаваемый объект доступен как this, так что можно проставить любые свойства.

В этом примере был создан объект класса Animal, и ему добавлены свойства name иcanWalk. Получилось вот так:

|  |  |
| --- | --- |
| animal.name = 'скотинка' | |
| animal.canWalk = true |

Класс объекта определяется функцией, которая его создала. Для проверки принадлежности классу есть оператор instanceof:

|  |
| --- |
| alert(animal instanceof Animal)  // => true |

Этот оператор иногда не работает, как полагается. Далее мы подробно разберем логику его работы, чтобы понимать, при каком наследовании и как его использовать.

## [Наследование через прототип](http://javascript.ru/tutorial/object/inheritance#nasledovanie-cherez-prototip)

В javascript базовое наследование основано не на классах. То есть, нет такого, что**классы наследуют друг от друга**, а объект класса-потомка получает общие свойства.

Вместо этого **объекты наследуют от объектов** без всяких классов. Наследование на классах можно построить(эмулировать), опираясь на базовое наследование javascript.

Разберем подробнее, что такое наследование от объектов и как оно работает.

### [Свойство prototype и прототип](http://javascript.ru/tutorial/object/inheritance#svoystvo-prototype-i-prototip)

Реализуется наследование через неявную(внутреннюю) ссылку одного объекта на другой, который называется его *прототипом* и в [спецификации](http://javascript.ru/ecma/part8#_Prototype_) обозначается[[prototype]]. Это свойство обычно скрыто от программиста.

Также существует свойство с похожим названием prototype (без квадратных скобок) - оно вспомогательное и указывает, откуда брать прототип при создании объекта.

Когда вы ставите функции Animal свойство Animal.prototype = XXX - вы этим декларируете: "все новые объекты класса Animal будут иметь прототип XXX".

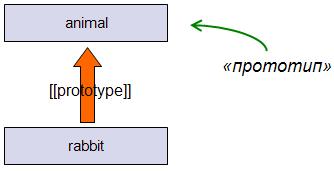
Исходя из [спецификации](http://javascript.ru/ecma) языка, ссылка на прототип объекта[[prototype]] не обязана быть доступной для чтения и изменения.

Однако в реализации javascript, используемой в Gecko-браузерах: Firefox/Mozilla и т.п., эта ссылка является обычным свойством объекта: \_\_proto\_\_.  
В этих браузерах ее можно читать и изменять.

Наследование происходит через скрытое свойство прототип [[prototype]], однако единственный кроссбраузерный способ указать прототип - это использовать свойство prototype функции-конструктора.

Например пусть объект кролик "rabbit" наследует от объекта животное "animal".

В наследовании на прототипах это реализуется как ссылка rabbit.[[prototype]] = animal:



Ссылка [[prototype]] работает так:

1. Любое запрошенное свойство ищется сначала в rabbit
2. Если свойство там не найдено, то оно ищется в rabbit.[[prototype]], т.е вanimal

Благодаря поиску по прототипу получается, что все функции и переменные, которые были в animal, доступны и в rabbit.

Ссылка на прототип создается оператором new во время создания объекта.

Ее значением становится свойство prototype функции-конструктора. Значение prototype указывает, от кого будут наследовать новые объекты

Прототип работает как резервное хранилище свойств. Если свойства нет у объекта - оно ищется в его прототипе. Получается наследование.

### [Пример](http://javascript.ru/tutorial/object/inheritance#primer)

Сделаем класс, наследующий от Animal - назовем его Rabbit.

Для этого сначала объявим функцию Rabbit.

|  |  |
| --- | --- |
| function Rabbit(name) { | |
| this.name = name |

|  |
| --- |
| } |

Пока что она просто создает объекты Rabbit. Поставим свойство prototype, чтобы новые объекты имели прототип animal (мы объявили этот объект чуть выше):

|  |
| --- |
| Rabbit.prototype = animal |

[Показать исходный код](http://javascript.ru/tutorial/object/inheritance#expandSource)

А теперь - создадим пару кроликов.

|  |  |
| --- | --- |
| 01 | big = new Rabbit('Chuk') |
| 02 | small = new Rabbit('Gek') | |

|  |  |
| --- | --- |
| 03 |  |
| 04 | alert(big.name)  // Chuk | |

|  |  |  |
| --- | --- | --- |
| 05 | alert(small.name) // Gek | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | alert(big.canWalk) // true | |
| 08 |  |

|  |  |
| --- | --- |
| 09 | // в Firefox можно еще так |
| 10 | if (big.\_\_proto\_\_) {  // в Firefox \_\_proto\_\_ это [[Prototype]] | |

|  |  |  |
| --- | --- | --- |
| 11 | alert(big.\_\_proto\_\_.name) // скотинка | |
| 12 | } |

Свойство name хранится прямо в объектах Rabbit, а canWalk берется из прототипаanimal.

Так как у обоих кроликов один прототип, то его изменение тут же отразится на обоих.

|  |  |  |
| --- | --- | --- |
| 1 | alert(big.canWalk)  // true | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | // поменяли в прототипе | |
| 4 | animal.canWalk = false |

|  |  |
| --- | --- |
| 5 |  |
| 6 | alert(big.canWalk)  // false | |

|  |  |
| --- | --- |
| 7 | alert(small.canWalk)  // false |

#### [**Перекрытие свойств родителя**](http://javascript.ru/tutorial/object/inheritance#perekrytie-svoystv-roditelya)

Запишем свойство canWalk напрямую в объект Rabbit:

|  |  |  |
| --- | --- | --- |
| 1 | animal.canWalk = false | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | small.canWalk = true | |
| 4 |  |

|  |  |
| --- | --- |
| 5 | alert(big.canWalk)  // false |
| 6 | alert(small.canWalk)  // true | |

У разных кроликов получилось разное значение canWalk, независимое от родителя.

Таким образом мы реализовали *перекрытие* (override) свойств родительского объекта.

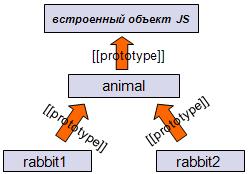
### [Начало цепочки наследования](http://javascript.ru/tutorial/object/inheritance#nachalo-cepochki-nasledovaniya)

Наверху цепочки всегда находится объект встроенного класса Object.

Так получается из-за того, что по умолчанию свойство prototype функции равно пустому объекту new Object().

|  |  |
| --- | --- |
| // Animal.prototype не указан явно, по умолчанию: | |
| Animal.prototype = {} |

Получается такая картинка:



Это хорошо, потому что у класса Object есть ряд полезных функций: toString(),hasOwnProperty()... А, например в Firefox, есть даже функция toSource(), которая дает исходный код, т.е "полный дамп" объекта.

Благодаря тому, что вверху цепочки наследования стоит Object, все остальные объекты имеют доступ к этому функционалу.

### [Методы объекта](http://javascript.ru/tutorial/object/inheritance#metody-obekta)

При вызове метода - он имеет доступ ко всем данным "своего" объекта.

Для этого в javascript (как, впрочем, и во многих других языках) используется ключевое слово this.

Например мы хотим добавить всем объектам класса Animal функцию перемещения. Для этого запишем в Animal.prototype метод move. Каждый его вызов будет изменять расстояние distance:

|  |  |  |
| --- | --- | --- |
| 1 | Animal.prototype.move = function(n) { | |
| 2 | this.distance = n |

|  |  |  |
| --- | --- | --- |
| 3 | alert(this.distance) | |
| 4 | } |

Теперь если мы сделаем новый объект, то он сможет передвигаться:

|  |  |  |
| --- | --- | --- |
| 1 | var animal = new Animal("животное") | |
| 2 | animal.move(3)   // => 3 |

|  |  |  |
| --- | --- | --- |
| 3 | animal.move(4)   // => 4 | |
| 4 | ... |

При вызове animal.move, интерпретатор находит нужный метод в прототипе animal:Animal.prototype.move и выполняет его, устанавливая this в "текущий" объект.

*this в javascript*

В javascript this работает не так, как в PHP, C, Java.

Значение this ставится на этапе вызова функции и может быть различным, в зависимости от контекста.

Подробнее это описано в статье [как javascript работает с this](http://javascript.ru/tutorial/object/thiskeyword).

Точно также смогут вызывать move и объекты класса Rabbit, так как их прототипом является animal.

Методы класса объявляются в Класс.prototype. Например,Animal.prototype - содержит методы для всех объектов класса Animal.

Альтернативный подход заключается в добавлении методов объекту в его конструкторе.

Объявление move в классе Animal при таком подходе выглядело бы вот так:

|  |  |
| --- | --- |
| 1 | function Animal(n) { |
| 2 | // конструируем объект | |

|  |  |
| --- | --- |
| 3 | ..... |
| 4 | // добавляем методы | |

|  |  |  |
| --- | --- | --- |
| 5 | this.move = function(n) { | |
| 6 | this.distance = n |

|  |  |  |
| --- | --- | --- |
| 7 | alert(this.distance) | |
| 8 | } |

|  |  |
| --- | --- |
| 9 | } |

В наиболее распространенных javascript-библиотеках используется первый подход, т.е добавление методов в прототип.

*Свойства-объекты или "иногда прототип это зло"*

Объявление всех свойств в прототипе может привести к незапланированному разделению одного и того же свойства разными объектами.

Например, объявим объект класса хомяк(Hamster). Метод foundнабирает еду за щеки, набранное хранит в массиве food.

|  |  |  |
| --- | --- | --- |
| 1 | function Hamster() {  } | |
| 2 | Hamster.prototype = { |

|  |  |
| --- | --- |
| 3 | food: [], |
| 4 | found: function(something) { | |

|  |  |  |
| --- | --- | --- |
| 5 | this.food.push(something) | |
| 6 | } |

|  |  |
| --- | --- |
| 7 | } |

Создадим двух хомячков: speedy и lazy и накормим первого:

|  |  |  |
| --- | --- | --- |
| 1 | speedy = new Hamster() | |
| 2 | lazy = new Hamster() |

|  |  |
| --- | --- |
| 3 |  |
| 4 | speedy.found("apple") | |

|  |  |  |
| --- | --- | --- |
| 5 | speedy.found("orange") | |
| 6 |  |

|  |  |
| --- | --- |
| 7 | alert(speedy.food.length) // 2 |
| 8 | alert(lazy.food.length) // 2 (!??) | |

[Открыть этот код в новом окне](http://javascript.ru/files/upload/object/hamster.html)

Как видно - второй хомяк тоже оказался накормленным! В чем дело?

Причина заключается в том, что food не является элементарным значением.

Если при простом присвоении hamster.property="..." меняется свойство property непосредственно в объекте hamster, то при вызовеhamster.food.push(...) - яваскрипт сначала находит свойство food - а так, как в hamster его нет, то оно берется из прототипаHamster.prototype, а затем вызывает для него метод push.

На каком бы хомяке не вызывался hamster.food.push(..) - свойствоfood будет браться одно и то же, из общего прототипа всех хомяков.

Мы получили пример "статического свойства класса". Да, оно бывает полезно. Например, мы можем разделять общую информацию между всеми хомяками посетителя.

Но в данном случае такое ни к чему.

Чтобы разделить данные, неэлементарные свойства обычно присваивают в конструкторе:

|  |  |  |
| --- | --- | --- |
| 1 | function Hamster() { | |
| 2 | this.food = [] |

|  |  |
| --- | --- |
| 3 | } |
| 4 | Hamster.prototype = { | |

|  |  |  |
| --- | --- | --- |
| 5 | food: [], // просто для информации | |
| 6 | found: function(something) { |

|  |  |  |
| --- | --- | --- |
| 7 | this.food.push(something) | |
| 8 | } |

|  |  |
| --- | --- |
| 9 | } |

Теперь у каждого объекта-хомяка будет свой собственный массивfood.

Свойство food в прототипе оставлено как комментарий. Оно не используется, но может быть полезно для удобства документирования.

## [Наследование на классах. Функция extend](http://javascript.ru/tutorial/object/inheritance#nasledovanie-na-klassah-funkciya-extend)

Рабочий вариант наследования на классах, в общем-то, готов.

Для того чтобы объект класса Rabbit унаследовал от класса Animal - нужно

1. Описать Animal
2. Описать Rabbit
3. Унаследовать кролика от объекта Animal:

|  |
| --- |
| Rabbit.prototype = new Animal() |

Однако, у такого подхода есть два недостатка:

1. Для наследования создается совершенно лишний объект new Animal()
2. Конструктор Animal должен предусматривать этот лишний вызов для и при необходимости делать такое "недоживотное", годное лишь на прототип.

К счастью, можно написать такую функцию, которая будет брать два класса и делать первый потомком второго:

|  |  |  |
| --- | --- | --- |
| 1 | function extend(Child, Parent) { | |
| 2 | var F = function() { } |

|  |  |  |
| --- | --- | --- |
| 3 | F.prototype = Parent.prototype | |
| 4 | Child.prototype = new F() |

|  |  |
| --- | --- |
| 5 | Child.prototype.constructor = Child |
| 6 | Child.superclass = Parent.prototype |

|  |  |
| --- | --- |
| 7 | } |

Использовать ее для наследования можно так:

|  |  |
| --- | --- |
| 01 | // создали базовый класс |
| 02 | function Animal(..) { ... } | |

|  |  |
| --- | --- |
| 03 |  |
| 04 | // создали класс | |

|  |  |  |
| --- | --- | --- |
| 05 | // и сделали его потомком базового | |
| 06 | function Rabbit(..)  { ... } |

|  |  |  |
| --- | --- | --- |
| 07 | extend(Rabbit, Animal) | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | // добавили в класс Rabbit методы и свойства | |
| 10 | Rabbit.prototype.run = function(..) { ... } |

|  |  |
| --- | --- |
| 11 |  |
| 12 | // все, теперь можно создавать объекты | |

|  |  |  |
| --- | --- | --- |
| 13 | // класса-потомка и использовать методы класса-родителя | |
| 14 | rabbit = new Rabbit(..) |

|  |  |
| --- | --- |
| 15 | rabbit.animalMethod() |

Функция очень удобная и работает "на ура".

Она не создает лишних объектов и в качестве бонуса записывает класс-родитель в свойство потомка superclass - это удобно для вызова родительских методов в конструкторе и при перекрытии методов.

*Как оно работает?*

Есть разные мнения, кто придумал функцию extend, но популяризацией она обязана [Дугласу Крокфорду](http://www.crockford.com/).

Как и почему она все-таки работает - может быть неочевидно даже опытным javascript-специалистам.

Попробуйте понять это, полистав [спецификацию](http://javascript.ru/ecma), особенно параграфы 13.2 и 15.3, а если какие-то вопросы остались - читайте дальше.

Предупреждение. Объяснение сложное, подробное и, вообще говоря, не обязательное, ведь функция "просто работает". Читайте на свой страх и риск.

### [Первая строка. Вспомогательный объект F](http://javascript.ru/tutorial/object/inheritance#pervaya-stroka-vspomogatelnyy-obekt-f)

Здесь мы разберем то, что, вообще говоря, происходит при создании любой функции (и в первой строке extend).

mwsnap023.jpg

Этот синтаксис - ни что иное как удобная форма записи для:

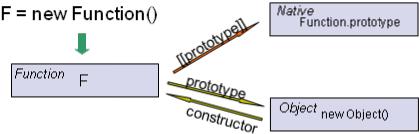
|  |
| --- |
| F = new Function() |

Эта строка cоздает новый объект класса Function (встроенный класс javascript).

Конструктор Function хранит ссылку на Function.prototype, который содержит общие свойства функций: call, apply, constructor, toString и т.п. Поэтому F.[[prototype]] = Function.prototype.

Кстати, за счет такого прототипа все функции и имеют доступ к методам call, applyи т.д.

Создание объекта F можно изобразить так:



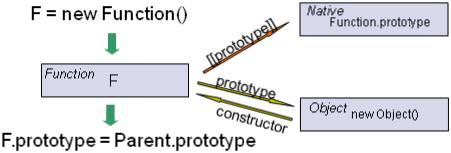
На картинке также изображено свойство prototype, которое автоматически устанавливается в new Object(). Свойство constructor также генерируется интерпретатором и показывает обратно, так что по кругу prototype.constructor для функции можно идти бесконечно: F.prototype.constructor === F.

### [Вторая строка. Меняем F.prototype](http://javascript.ru/tutorial/object/inheritance#vtoraya-stroka-menyaem-f-prototype)

Следующая строка устанавливает свойство F.prototype:

|  |
| --- |
| F.prototype = Parent.prototype |

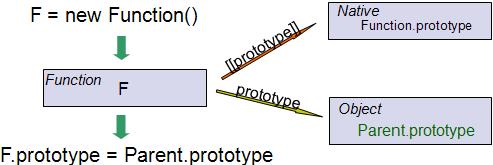
До второй строки свойства имели такие значения:



Свойство F.prototype указывало на объект new Object() (справа снизу на рисунке).

Теперь значение поменялось, и старый new Object() перестал быть доступен - ни одна ссылка на него не ведет. Поэтому сборщик мусора убивает его.

Вот так изменения отразятся на картине:



### [Третья строка. Prototype для дочернего класса](http://javascript.ru/tutorial/object/inheritance#tretya-stroka-prototype-dlya-dochernego-klassa)

Следующая строка устанавливает свойство prototype для дочернего класса, чтобы оно служило прототипом всех дочерних объектов.

|  |
| --- |
| Child.prototype = new F() |

При создании объекта класса F, свойство [[prototype]] нового объекта конструктор возьмет из F.prototype:

|  |
| --- |
| (new F).[[prototype]] = (т.к F.prototype==Parent.prototype) = Parent.prototype |

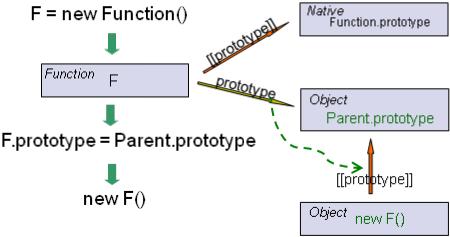
т.е получится такая цепочка присвоения

|  |
| --- |
| Child.prototype.[[prototype]] = (new F).[[prototype]] = Parent.prototype |

Иначе говоря, у нас получилось, что

|  |
| --- |
| Child.prototype = [объект,  прототип которого - Parent.prototype] |

Это присвоение можно изобразить на картинке вот так:



В правом-нижнем углу - как раз и находится Child.prototype, прототипом которого получился Parent.prototype.

Собственно, наследование уже работает. В самом деле, создадим новый объект классаChild:

|  |
| --- |
| child = new Child(...) |

Вспоминаем, что интерпретатор, выполняя new, ставит child.[[prototype]] = Child.prototype.

Поиск свойств, не найденных в child будет идти по  
child.[[prototype] = Child.prototype = new F().  
Если там не нашли, то интерпретатор будет искать в new F().[[prototype]] = F.prototype = Parent.prototype, то есть, в конечном счете:

|  |
| --- |
| child -> Child.prototype -> Parent.prototype |

.. Что и требовалось получить.

### [Четвертая строка. Поправить свойство constructor](http://javascript.ru/tutorial/object/inheritance#chetvertaya-stroka-popravit-svoystvo-constructor)

Свойство Child.prototype.constructor осталось старое, и его нужно поправить строкой:

|  |
| --- |
| Child.prototype.constructor = Child |

Для каждой функции свойство prototype.constructor всегда должно указывать на саму функцию.

Если это так, то объекты, которые функция создает, тоже будут иметь (через прототип) правильное свойство constructor, указывающее на создавшую их функцию.

Например:

|  |  |
| --- | --- |
| 1 | function Z() {} |
| 2 | alert(Z.prototype.constructor)  // => функция Z | |

|  |  |
| --- | --- |
| 3 |  |
| 4 | z = new Z() | |

|  |  |  |
| --- | --- | --- |
| 5 | //в z нет ничего, но в z.[[prototype]]=Z.prototype есть constructor | |
| 6 | alert(z.constructor)   // => функция Z |

Эта строка extend как раз и проставляет правильное свойство prototype.constructor.

### [Пятая строка. Сохранить суперкласс](http://javascript.ru/tutorial/object/inheritance#pyataya-stroka-sohranit-superklass)

Добавим в класс явную ссылку на родительский класс для удобного обращения к его методам. Понадобится для вызова конструктора родителя или если родительский метод был перекрыт в потомке.

|  |
| --- |
| Child.superclass = Parent.prototype |

## [Вызов родительских методов](http://javascript.ru/tutorial/object/inheritance#vyzov-roditelskih-metodov)

### [Конструктор](http://javascript.ru/tutorial/object/inheritance#konstruktor)

В механизме наследования, разобранном выше, есть одно белое пятно. Это - конструктор.

Хотелось бы, чтобы конструкторы всех родителей вызывались по порядку до конструктора самого объекта.

С наследованием через extend - это очень просто.

Вызов конструктора родителя с теми же аргументами, что были переданы осуществляется так:

|  |  |  |
| --- | --- | --- |
| 1 | function Rabbit(..)  { | |
| 2 | ... |

|  |  |  |
| --- | --- | --- |
| 3 | Rabbit.superclass.constructor.apply(this, arguments) | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | ... | |
| 6 | } |

Конечно же, аргументы можно поменять, благо *apply* дает возможность вызвать функцию с любыми параметрами вместо arguments в примере.

### [Любые методы](http://javascript.ru/tutorial/object/inheritance#lyubye-metody)

Аналогично можно вызвать и любой другой метод родительского класса:

|  |
| --- |
| Rabbit.superclass.run.apply(this, ...) |

### [Почему не this.constructor?](http://javascript.ru/tutorial/object/inheritance#pochemu-ne-this-constructor)

В этих примерах везде в явном виде указано имя класса Rabbit, хотя можно бы попробовать указать this.constructor, который должен указывать на Rabbit, т.к объект принадлежит этому классу.

Если так поступить, то будет ошибка при цепочке наследования классов из 3 элементов типа foo -> bar -> zot.

Проиллюстрируем ее на примере:

[Показать исходный код](http://javascript.ru/tutorial/object/inheritance#expandSource)

Последний вызов приведет к ошибке "too much recursion" из-за того, чтоthis.constructor.superclass, к которому идет обращение в функции bar.identityобращается к дочернему классу zot. В результате bar.identity вызывает сама себя в бесконечной рекурсии.

Правильный способ заключается в явном обозначении класса, т.еRabbit.superclass...

### [Оператор instanceOf](http://javascript.ru/tutorial/object/inheritance#operator-instanceof)

Оператор instanceOf проверяет принадлежность объекта классу, проходя по цепочке его прототипов, и используя для сравнения свойство prototype.

Логику его работы можно описать так:

|  |  |  |
| --- | --- | --- |
| 1 | function instanceOf(object, constructor) { | |
| 2 | var o=object |

|  |  |
| --- | --- |
| 3 |  |
| 4 | while (o.\_\_proto\_\_ != null) { | |

|  |  |  |
| --- | --- | --- |
| 5 | if (o.\_\_proto\_\_ === constructor.prototype) return true | |
| 6 | o = o.\_\_proto\_\_ |

|  |  |
| --- | --- |
| 7 | } |
| 8 | return false | |

|  |  |
| --- | --- |
| 9 | } |

Поэтому при правильной структуре прототипов он всегда корректно работает.

У этого оператора есть неприятная особенность при использовании нескольких окон: в разных окнах объекты классов (окружение) разное, поэтому массив из одного окна(фрейма) не будет опознан как Array в другом фрейме.

Впрочем, такая ситуация возникает довольно редко.

## [Полный пример наследования](http://javascript.ru/tutorial/object/inheritance#polnyy-primer-nasledovaniya)

Для окончательной организации удобного javascript-наследования на классе, пригодится функция копирования свойств из объекта src в другой dst:

|  |  |
| --- | --- |
| 01 | // копирует все свойства из src в dst, |
| 02 | // включая те, что в цепочке прототипов src до Object | |

|  |  |
| --- | --- |
| 03 | function mixin(dst, src){ |
| 04 | // tobj - вспомогательный объект для фильтрации свойств, | |

|  |  |  |
| --- | --- | --- |
| 05 | // которые есть у объекта Object и его прототипа | |
| 06 | var tobj = {} |

|  |  |
| --- | --- |
| 07 | for(var x in src){ |
| 08 | // копируем в dst свойства src, кроме тех, которые унаследованы от Object | |

|  |  |  |
| --- | --- | --- |
| 09 | if((typeof tobj[x] == "undefined") || (tobj[x] != src[x])){ | |
| 10 | dst[x] = src[x]; |

|  |  |  |
| --- | --- | --- |
| 11 | } | |
| 12 | } |

|  |  |  |
| --- | --- | --- |
| 13 | // В IE пользовательский метод toString отсутствует в for..in | |
| 14 | if(document.all && !document.isOpera){ |

|  |  |
| --- | --- |
| 15 | var p = src.toString; |
| 16 | if(typeof p == "function" && p != dst.toString && p != tobj.toString && | |

|  |  |  |
| --- | --- | --- |
| 17 | p != "\nfunction toString() {\n    [native code]\n}\n"){ | |
| 18 | dst.toString = src.toString; |

|  |  |  |
| --- | --- | --- |
| 19 | } | |
| 20 | } |

|  |  |
| --- | --- |
| 21 | } |

В полном примере мы создадим класс Animal c методом walk и его насленика Bird, который умеет летать: fly. Функции walk и fly принимают время ходьбы/полета и соответственно увеличивают свойство distance - расстояние до животного:

|  |  |  |
| --- | --- | --- |
| 01 | // ---- родительский класс ---- | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | function Animal(name, walkSpeed) { | |
| 04 | this.name = name |

|  |  |  |
| --- | --- | --- |
| 05 | this.walkSpeed = walkSpeed | |
| 06 | } |

|  |  |
| --- | --- |
| 07 |  |
| 08 | // добавляем методы объекта | |

|  |  |  |
| --- | --- | --- |
| 09 | mixin(Animal.prototype, { | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | // пример переменной | |
| 12 | distance: 0, |

|  |  |
| --- | --- |
| 13 |  |
| 14 | // пример метода | |

|  |  |
| --- | --- |
| 15 | walk: function(time) { |
| 16 | this.distance = this.distance + time\*this.walkSpeed | |

|  |  |  |
| --- | --- | --- |
| 17 | }, | |
| 18 |  |

|  |  |
| --- | --- |
| 19 | toString: function() { |
| 20 | return this.name+" на расстоянии "+this.distance | |

|  |  |  |
| --- | --- | --- |
| 21 | } | |
| 22 | }) |

|  |  |
| --- | --- |
| 23 |  |
| 24 | // ---- класс наследник ---- | |

|  |  |
| --- | --- |
| 25 |  |
| 26 | function Bird(name, walkSpeed, flySpeed) { | |

|  |  |
| --- | --- |
| 27 | // вызов родительского конструктора |
| 28 | Bird.superclass.constructor.call(this, name, walkSpeed) | |

|  |  |
| --- | --- |
| 29 |  |
| 30 | this.flySpeed = flySpeed | |

|  |  |
| --- | --- |
| 31 | } |
| 32 | extend(Bird, Animal) | |

|  |  |
| --- | --- |
| 33 |  |
| 34 | mixin(Bird.prototype, { | |

|  |  |
| --- | --- |
| 35 | fly: function(time) { |
| 36 | this.distance = this.distance + time\*this.flySpeed | |

|  |  |  |
| --- | --- | --- |
| 37 | } | |
| 38 | }) |

Пример создания объекта-наследника:

|  |  |  |
| --- | --- | --- |
| 1 | bird = new Bird("Птыц", 1, 10) | |
| 2 |  |

|  |  |  |
| --- | --- | --- |
| 3 | bird.walk(3) | |
| 4 |  |

|  |  |  |
| --- | --- | --- |
| 5 | alert(bird) // => Птыц на расстоянии 3 | |
| 6 |  |

|  |  |  |
| --- | --- | --- |
| 7 | bird.fly(2) | |
| 8 |  |

|  |  |
| --- | --- |
| 9 | alert(bird) // => Птыц на расстоянии 23 |

Конечно же, вызовы extend и mixin можно объединить в одну функцию. В примере это не сделано для наглядности происходящего.

### [Private члены](http://javascript.ru/tutorial/object/inheritance#private-chleny)

При наследовании можно организовать "настоящие" приватные члены класса. Для этого, однако, придется объявлять все методы не отдельно от конструктора, а внутри него:

|  |  |  |
| --- | --- | --- |
| 01 | function extend(Child, Parent) { | |
| 02 | var F = function() { } |

|  |  |  |
| --- | --- | --- |
| 03 | F.prototype = Parent.prototype | |
| 04 | Child.prototype = new F() |

|  |  |
| --- | --- |
| 05 | Child.prototype.constructor = Child |
| 06 | Child.superclass = Parent.prototype |

|  |  |  |
| --- | --- | --- |
| 07 | } | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | // ---- родительский класс ---- | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | function Animal(name, walkSpeed) { | |
| 12 |  |

|  |  |  |
| --- | --- | --- |
| 13 | // объявить приватную переменную | |
| 14 | var speed = walkSpeed |

|  |  |
| --- | --- |
| 15 |  |
| 16 | // объявить открытую переменную | |

|  |  |  |
| --- | --- | --- |
| 17 | this.distance = 0 | |
| 18 |  |

|  |  |  |
| --- | --- | --- |
| 19 | // добавить метод, использующий private speed | |
| 20 | this.walk = function(time) { |

|  |  |  |
| --- | --- | --- |
| 21 | this.distance = this.distance + time\*speed | |
| 22 | } |

|  |  |
| --- | --- |
| 23 |  |
| 24 | // добавить метод, использующий private name | |

|  |  |
| --- | --- |
| 25 | this.toString = function() { |
| 26 | return name+" на расстоянии "+this.distance | |

|  |  |  |
| --- | --- | --- |
| 27 | } | |
| 28 | } |

|  |  |
| --- | --- |
| 29 |  |
| 30 |  |

|  |  |  |
| --- | --- | --- |
| 31 | // ---- класс наследник ---- | |
| 32 |  |

|  |  |  |
| --- | --- | --- |
| 33 | function Bird(name, walkSpeed, flySpeed) { | |
| 34 | // вызов родительского конструктора |

|  |  |  |
| --- | --- | --- |
| 35 | Bird.superclass.constructor.call(this, name, walkSpeed) | |
| 36 |  |

|  |  |
| --- | --- |
| 37 | this.fly = function(time) { |
| 38 | this.distance = this.distance + time\*flySpeed | |

|  |  |  |
| --- | --- | --- |
| 39 | } | |
| 40 | } |

|  |  |  |
| --- | --- | --- |
| 41 | extend(Bird, Animal) | |
| 42 |  |

|  |  |
| --- | --- |
| 43 |  |
| 44 | bird = new Bird("Птыц", 1, 10) | |

|  |  |
| --- | --- |
| 45 |  |
| 46 | bird.walk(3) | |

|  |  |
| --- | --- |
| 47 |  |
| 48 | alert(bird) // => Птыц на расстоянии 3 | |

|  |  |
| --- | --- |
| 49 |  |
| 50 | bird.fly(2) | |

|  |  |
| --- | --- |
| 51 |  |
| 52 | alert(bird) // => Птыц на расстоянии 23 | |

Приватными являются все свойства, которые доступны только из внутренних методов объекта через механизм замыкания [(см. статью о функциях javascript)](http://javascript.ru/basic/functions).

Это свойства, явно объявленные через var, плюс аргументы конструктора.

При таком способе объявления - все свойства и методы записываются не в прототип объекта, а в сам объект.

Поэтому, если объектов создается очень много, то это сопряжено с дополнительными расходами памяти на хранение множества копий методов - свой код функции для каждого объекта, а не один в прототипе на всех. Обычно же эти расходы можно во внимание не принимать.

### [Protected члены](http://javascript.ru/tutorial/object/inheritance#protected-chleny)

Если Вы использовали ООП в других языках программирования, то наверняка знаете, что чаще делаются не private свойства, а protected, т.е такие, к которым могут получить доступ наследники. Javascript не предоставляет синтаксиса для создания protected свойств, поэтому их просто помечают подчеркиванием в начале.

Например,

|  |  |  |
| --- | --- | --- |
| 01 | function Animal(name) { | |
| 02 |  |

|  |  |  |
| --- | --- | --- |
| 03 | var privateVariable = 0 | |
| 04 |  |

|  |  |  |
| --- | --- | --- |
| 05 | this.\_protectedName = name | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | this.\_protectedMethod = function(..) { | |
| 08 | ... alert(privateVariable).. |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 |  |

|  |  |  |
| --- | --- | --- |
| 11 | this.publicMethod = function() { ... } | |
| 12 | } |

Все функции, объявленные внутри конструктора, имеют доступ к приватным свойствам и, конечно же, к защищенным и публичным.

Ограничение доступа к таким "защищенным" свойствам не жесткое и остается на совести программиста.

## [Фабрика объектов (мой любимый способ)](http://javascript.ru/tutorial/object/inheritance#factory)

Есть альтернативный способ наследования, который вообще не требует вызова new.

При этом объекты создаются с помощью "фабричных функций" (factory function). Например, в следующем примере это функция Animal, которая производит некие действия и возвращает объект.

|  |  |  |
| --- | --- | --- |
| 01 | function Animal(name) { | |
| 02 | var speed = 10 |

|  |  |
| --- | --- |
| 03 | return { |
| 04 | name: name, | |

|  |  |
| --- | --- |
| 05 | run: function(distance) { |
| 06 | return distance / speed | |

|  |  |  |
| --- | --- | --- |
| 07 | } | |
| 08 | } |

|  |  |
| --- | --- |
| 09 | } |
| 10 | pet1 = Animal() | |

|  |  |
| --- | --- |
| 11 | pet2 = Animal() |

Через замыкания [(о функциях и замыканиях)](http://javascript.ru/basic/functions) организуются "приватные" члены класса. В примере выше доступ к переменной speed возможен только из функции run.

Задача фабрики объектов - создать объект и инициализировать его.

### [Создание потомка](http://javascript.ru/tutorial/object/inheritance#sozdanie-potomka)

Для создания потомка фабрика объектов просто модифицирует объект, создаваемой функцией-родителем.

Рассмотрим пример с созданием Rabbit - потомка Animal:

|  |  |  |
| --- | --- | --- |
| 01 | function Rabbit(name) { | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | // вызвать конструктор родителя, |
| 04 | // получить родительский объект в me | |

|  |  |  |
| --- | --- | --- |
| 05 | var me = Animal(name) | |
| 06 |  |

|  |  |  |
| --- | --- | --- |
| 07 | // добавить приватную переменную | |
| 08 | var jumps = 0 |

|  |  |
| --- | --- |
| 09 |  |
| 10 | /\* добавить новые методы к me \*/ | |

|  |  |
| --- | --- |
| 11 | me.jump = function() { jumps++ } |
| 12 | me.getJumps = function() { return jumps } | |

|  |  |
| --- | --- |
| 13 |  |
| 14 | // поставить правильное свойство конструктора | |

|  |  |  |
| --- | --- | --- |
| 15 | // (делаем вид, что объект создали мы, а не Animal) | |
| 16 | me.constructor = arguments.callee |

|  |  |
| --- | --- |
| 17 |  |
| 18 | return me | |

|  |  |
| --- | --- |
| 19 | } |

При создании потомка фабричная функция делает следующее:

1. Создает объект родительского класса
2. Присваивает ему публичные свойства и методы
3. Меняет свойство constructor объекта на себя
   * этот шаг можно пропустить (см дальше)

Кроме того, при необходимости через var объявляются собственные приватные члены, к которым будут иметь доступы все функции, объявленные внутри фабричной.

Фактически, функция берет другой объект и добавляет ему свои методы. Из-за этого такую реализацию наследования иногда называют**паразитическим наследованием**.

Почему этот способ мой любимый?

Во-первых, потому, что именно такой стиль ООП, как мне кажется, наиболее соответствует по духу яваскрипт.

Он прост и понятен.

В нем есть приватные переменные, которые замечательно сжимаются [javascript-минификатором](http://javascript.ru/optimize/javascript-compression), что существенно сокращает объем скачиваемого посетителем кода и увеличивает производительность, т.к интерпретатор javascript быстрее работает с короткими именами переменных.

### [instanceof ?](http://javascript.ru/tutorial/object/inheritance#instanceof)

К сожалению, метод instanceof при таком наследовании не работает, так как он завязан на [[prototype]] объекта. Фактически, вызов instanceof A просто проверяет, есть ли в цепочке прототипов объекта класс A.

Свойство constructor, которое было присвоено на 3м шаге, как раз и служит для эмуляции instanceof:

|  |
| --- |
| alert(rabbit.constructor === Rabbit) // => true |

Более сложная реализация наследования не присваивает constructor, а добавляет его в специальный список. Функция-аналог instanceofпроверяет всю цепочку и выдает, есть ли в ней искомый класс.

При таком наследовании часто применяют альтернативный подход кinstanceOf: проверку на нужный метод.

Как часто описывают такой способ - ".. Если объект умеет крякать, значит это утка. Кому какое дело что он на самом деле..":

|  |  |
| --- | --- |
| 1 | if (arr.splice) { |
| 2 | // умеет splice, значит это массив | |

|  |  |  |
| --- | --- | --- |
| 3 | // вообще, какая разница, что это за объект на самом деле, | |
| 4 | // то, что надо, он умеет - пользуем.. |

|  |  |  |
| --- | --- | --- |
| 5 | .. arr.splice(..) .. | |
| 6 | } |

### [Вызов свойств родителя](http://javascript.ru/tutorial/object/inheritance#vyzov-svoystv-roditelya)

В наследовании через классы свойства родителя доступны через superclass.

Здесь - чтобы получить доступ к методу родительского объекта, его обычно копируют куда-нибудь в замыкание. Например:

|  |  |
| --- | --- |
| 01 | function Rabbit(name) { |
| 02 | var me = animal(name) | |

|  |  |
| --- | --- |
| 03 |  |
| 04 | var jumps = 0 | |

|  |  |
| --- | --- |
| 05 |  |
| 06 | me.jump = function() { jumps++ } | |

|  |  |  |
| --- | --- | --- |
| 07 | me.getJumps = function() { return jumps } | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | // скопировать метод run родителя в замыкание | |
| 10 | var super\_run = me.run |

|  |  |
| --- | --- |
| 11 |  |
| 12 | /\* перекрыть метод \*/ | |

|  |  |  |
| --- | --- | --- |
| 13 | me.run = function(distance) { | |
| 14 | this.jump() |

|  |  |
| --- | --- |
| 15 |  |
| 16 | // вызвать родительский метод | |

|  |  |  |
| --- | --- | --- |
| 17 | return super\_run.call(this, distance) | |
| 18 | } |

|  |  |  |
| --- | --- | --- |
| 19 | return me | |
| 20 | } |

Таким образом нужно сделать "бэкап" всех нужных свойств.

Пожалуй, все основные моменты разобрали. В общем, фабрика объектов - простой и удобный способ наследования, работающий принципиально по-другому, нежели классы.

4.Область видимости, Контекст, call, apply

#### **Что такое область видимости?**

В JS область видимости – это текущий контекст в коде. ОВ могут быть определены локально или глобально. Ключ к написанию пуленепробиваемого кода – понимание ОВ. Давайте разбираться, где переменные и функции доступны, как менять контекст в коде и писать более быстрый и поддерживаемый код (который и отлаживать быстрее). Разбираться с ОВ просто – задаём себе вопрос, в какой из ОВ мы сейчас находимся, в А или в Б?

#### **Что есть глобальная/локальная ОВ?**

Не написав ни строчки кода, мы уже находимся в глобальной ОВ. Если мы сразу определяем переменную, она находится в глобальной ОВ.

// глобальная ОВ

var name = 'Todd';

Глобальная ОВ – ваш лучший друг и худший кошмар. Обучаясь работе с разными ОВ, вы не встретите проблем с глобальной ОВ, разве что вы увидите пересечения имён. Часто можно услышать «глобальная ОВ – это плохо», но нечасто можно получить объяснение – почему. ГОВ – не плохо, вам нужно её использовать при создании модулей и API, которые будут доступны из разных ОВ, просто нужно использовать её на пользу и аккуратно.  
  
Все мы использовали jQuery. Как только мы пишем

jQuery('.myClass');

мы получаем доступ к jQuery в глобальной ОВ, и мы можем назвать этот доступ пространством имён. Иногда термин «пространство имён» используют вместо термина ОВ, однако обычно им обозначают ОВ самого уровня. В нашем случае jQuery находится в глобальной ОВ, и является нашим пространством имён. Пространство имён jQuery определено в глобальной ОВ, которая работает как ПИ для библиотеки jQuery, в то время как всё её содержимое наследуется от этого ПИ.

#### **Что такое локальная ОВ?**

Локальной ОВ называют любую ОВ, определённую после глобальной. Обычно у нас есть одна ГОВ, и каждая определяемая функция несёт в себе локальную ОВ. Каждая функция, определённая внутри другой функции, имеет своё локальное ОВ, связанное с ОВ внешней функции.  
  
Если я определю функции и задам внутри переменные, они принадлежат локальной ОВ. Пример:

// ОВ A: глобальная

var myFunction = function () {

// ОВ B: локальная

};

Все переменные из ЛОВ не видны в ГОВ. К ним нельзя получить доступ снаружи напрямую. Пример:

var myFunction = function () {

var name = 'Todd';

console.log(name); // Todd

};

// ReferenceError: name is not defined

console.log(name);

Переменная “name” относится к локальной ОВ, она не видна снаружи и поэтому не определена. 

#### **Функциональная ОВ.**

Все локальные ОВ создаются только в функциональных ОВ, они не создаются циклами типа for или while или директивами типа if или switch. Новая функция – новая область видимости. Пример:

// ОВ A

var myFunction = function () {

// ОВ B

var myOtherFunction = function () {

// ОВ C

};

};

Так просто можно создать новую ОВ и локальные переменные, функции и объекты.

#### **Лексическая ОВ**

Если одна функция определена внутри другой, внутренняя имеет доступ к ОВ внешней. Это называется «лексической ОВ», или «замыканием», или ещё «статической ОВ». 

var myFunction = function () {

var name = 'Todd';

var myOtherFunction = function () {

console.log('My name is ' + name);

};

console.log(name);

myOtherFunction(); // вызов функции

};

// Выводит:

// `Todd`

// `My name is Todd`

С лексической ОВ довольно просто работать – всё, что определено в ОВ родителя, доступно в ОВ ребенка. К примеру:

var name = 'Todd';

var scope1 = function () {

// name доступно здесь

var scope2 = function () {

// name и здесь

var scope3 = function () {

// name и даже здесь!

};

};

};

В обратную сторону это не работает:

// name = undefined

var scope1 = function () {

// name = undefined

var scope2 = function () {

// name = undefined

var scope3 = function () {

var name = 'Todd'; // локальная ОВ

};

};

};

Всегда можно вернуть ссылку на “name”, но не саму переменную.

#### **Последовательности ОВ**

Последовательности ОВ определяют ОВ любой выбранной функции. У каждой определяемой функции есть своя ОВ, и каждая функция, определяемая внутри другой, имеет свой ОВ, связанный с ОВ внешней – это и есть последовательность, или цепочка. Позиция в коде определяет ОВ. Определяя значение переменной, JS идёт от самой глубоко вложенной ОВ наружу, пока не найдёт искомую функцию, объект или переменную.

#### **Замыкания**

Живут в тесном союзе с лексическими ОВ. Хорошим примером использования является возврат ссылки на функцию. Мы можем возвращать наружу разные ссылки, которые делают возможным доступ к тому, что было определено внутри.

var sayHello = function (name) {

var text = 'Hello, ' + name;

return function () {

console.log(text);

};

};

Чтобы вывести на экран текст, недостаточно просто вызвать функцию sayHello:

sayHello('Todd'); // тишина

Функция возвращает функцию, поэтому её надо сначала присвоить, а потом вызвать:

var helloTodd = sayHello('Todd');

helloTodd(); // вызывает замыкание и выводит 'Hello, Todd'

Можно конечно вызвать замыкание и напрямую:

sayHello('Bob')(); // вызывает замыкание без присваивания

В AngularJS используются подобные вызовы в методеs $compile, где нужно передавать ссылку на текущую ОВ:

$compile(template)(scope);

Можно догадаться, что упрощённо их код выглядит примерно так:

var $compile = function (template) {

// всякая магия

// без доступа к scope

return function (scope) {

// здесь есть доступ и к `template` и к `scope`

};

};

Функция не обязана ничего возвращать, чтобы быть замыканием. Любой доступ к переменным извне текущей ОВ создаёт замыкание.

#### **ОВ и 'this'**

Каждая ОВ назначает своё значение переменной “this”, в зависимости от способа вызова функции. Мы все использовали ключевое слово this, но не все понимают, как оно работает и какие есть отличия при вызовах. По умолчанию, оно относится к объекту самой внешней ОВ, текущему окну. Пример того, как разные вызовы меняют значения this:

var myFunction = function () {

console.log(this); // this = глобальное, [объект Window]

};

myFunction();

var myObject = {};

myObject.myMethod = function () {

console.log(this); // this = текущий объект { myObject }

};

var nav = document.querySelector('.nav'); // <nav class="nav">

var toggleNav = function () {

console.log(this); // this = элемент <nav>

};

nav.addEventListener('click', toggleNav, false);

Встречаются и проблемы со значением this. В следующем примере внутри одной и той же функции значение и ОВ могут меняться:

var nav = document.querySelector('.nav'); // <nav class="nav">

var toggleNav = function () {

console.log(this); // <nav> element

setTimeout(function () {

console.log(this); // [объект Window]

}, 1000);

};

nav.addEventListener('click', toggleNav, false);

Здесь мы создали новую ОВ, которая вызывается не из обработчика событий, а значит, относится к объекту window. Можно, например, запоминать значение this в другой переменной, чтобы не возникало путаницы:

var nav = document.querySelector('.nav'); // <nav class="nav">

var toggleNav = function () {

var that = this;

console.log(that); // элемент <nav>

setTimeout(function () {

console.log(that); // элемент <nav>

}, 1000);

};

nav.addEventListener('click', toggleNav, false);

#### **Меняем ОВ при помощи .call(), .apply() и .bind()**

Иногда есть необходимость менять ОВ в зависимости от того, что вам нужно.   
В примере:

var links = document.querySelectorAll('nav li');

for (var i = 0; i < links.length; i++) {

console.log(this); // [объект Window]

}

Значение this не относится к перебираемым элементам, мы ничего не вызываем и не меняем ОВ. Давайте посмотрим, как мы можем менять ОВ (точнее, мы меняем контекст вызова функций).

#### **.call() and .apply()**

Методы .call() и .apply() позволяют передавать ОВ в функцию:

var links = document.querySelectorAll('nav li');

for (var i = 0; i < links.length; i++) {

(function () {

console.log(this);

}).call(links[i]);

}

В результате в this передаются значения перебираемых элементов. Метод .call(scope, arg1, arg2, arg3) принимает список аргументов, разделённых запятыми, а метод .apply(scope, [arg1, arg2]) принимает массив аргументов.  
  
Важно помнить, что методы .call() или .apply() вызывают функции, поэтому вместо

myFunction(); // вызывает myFunction

позвольте .call() вызвать функцию и передать параметр:

myFunction.call(scope);

#### **.bind()**

.bind() не вызывает функцию, а просто привязывает значения переменных перед её вызовом. Как вы знаете, мы не можем передавать параметры в ссылки на функции:

// работает

nav.addEventListener('click', toggleNav, false);

// приводит к немедленному вызову функции

nav.addEventListener('click', toggleNav(arg1, arg2), false);

Это можно исправить, создав новую вложенную функцию:

nav.addEventListener('click', function () {

toggleNav(arg1, arg2);

}, false);

Но тут опять происходит изменение ОВ, создание лишней функции, что негативно отразится на быстродействии. Поэтому мы используем .bind(), в результате мы можем передавать аргументы так, чтобы не происходило вызова функции:

nav.addEventListener('click', toggleNav.bind(scope, arg1, arg2), false);

5. Замыкания, самовызывающиеся функции, Инкапсуляция в Javascript

#### **Замыкания**

Живут в тесном союзе с лексическими ОВ. Хорошим примером использования является возврат ссылки на функцию. Мы можем возвращать наружу разные ссылки, которые делают возможным доступ к тому, что было определено внутри.

var sayHello = function (name) {

var text = 'Hello, ' + name;

return function () {

console.log(text);

};

};

Чтобы вывести на экран текст, недостаточно просто вызвать функцию sayHello:

sayHello('Todd'); // тишина

Функция возвращает функцию, поэтому её надо сначала присвоить, а потом вызвать:

var helloTodd = sayHello('Todd');

helloTodd(); // вызывает замыкание и выводит 'Hello, Todd'

Можно конечно вызвать замыкание и напрямую:

sayHello('Bob')(); // вызывает замыкание без присваивания

В AngularJS используются подобные вызовы в методеs $compile, где нужно передавать ссылку на текущую ОВ:

$compile(template)(scope);

Можно догадаться, что упрощённо их код выглядит примерно так:

var $compile = function (template) {

// всякая магия

// без доступа к scope

return function (scope) {

// здесь есть доступ и к `template` и к `scope`

};

};

Функция не обязана ничего возвращать, чтобы быть замыканием. Любой доступ к переменным извне текущей ОВ создаёт замыкание.

#### **Приватные и публичные ОВ**

В JavaScript, в отличии от многих других языков, нет понятий публичных и приватных ОВ, но мы можем их эмулировать при помощи замыканий. Для создания приватной ОВ мы можем обернуть наши функции в другие функции.

(function () {

// здесь приватная ОВ

})();

Добавим функциональности:

(function () {

var myFunction = function () {

// делаем здесь, что нужно

};

})();

Но вызвать эту функцию напрямую нельзя:

(function () {

var myFunction = function () {

// делаем здесь, что нужно

};

})();

myFunction(); // Uncaught ReferenceError: myFunction is not defined

Вот вам и приватная ОВ. Если вам нужна публичная ОВ, воспользуемся следующим трюком. Создаём пространство имён Module, которое содержит всё, относящееся к данному модулю:

// определяем модуль

var Module = (function () {

return {

myMethod: function () {

console.log('myMethod has been called.');

}

};

})();

// вызов методов модуля

Module.myMethod();

Директива return возвращает методы, доступные публично, в глобальной ОВ. При этом они относятся к нужному пространству имён. Модуль Module может содержать столько методов, сколько нужно.

// определяем модуль

var Module = (function () {

return {

myMethod: function () {

},

someOtherMethod: function () {

}

};

})();

// вызов методов модуля

Module.myMethod();

Module.someOtherMethod();

Не нужно стараться вываливать все методы в глобальную ОВ и загрязнять её. Вот так можно организовать приватную ОВ, не возвращая функции:

var Module = (function () {

var privateMethod = function () {

};

return {

publicMethod: function () {

}

};

})();

Мы можем вызвать publicMethod, но не можем privateMethod – он относится к приватной ОВ. В эти функции можно засунуть всё что угодно — addClass, removeClass, вызовы Ajax/XHR, Array, Object, и т.п.  
  
Интересный поворот в том, что внутри одной ОВ все функции имеют доступ к любым другим, поэтому из публичных методов мы можем вызывать приватные, которые в глобальной ОВ недоступны:

var Module = (function () {

var privateMethod = function () {

};

return {

publicMethod: function () {

// есть доступ к методу `privateMethod`:

// privateMethod();

}

};

})();

Это повышает интерактивность и безопасность кода. Ради безопасности не стоит вываливать все функции в глобальную ОВ, чтобы функции, которые вызывать не нужно, не вызвали бы ненароком.  
  
Пример возврата объекта с использованием приватных и публичных методов:

var Module = (function () {

var myModule = {};

var privateMethod = function () {

};

myModule.publicMethod = function () {

};

myModule.anotherPublicMethod = function () {

};

return myModule; // returns the Object with public methods

})();

// использование

Module.publicMethod();

Удобно начинать название приватных методов с подчёркивания, чтобы визуально отличать их от публичных:

var Module = (function () {

var \_privateMethod = function () {

};

var publicMethod = function () {

};

})();

Удобно также возвращать методы списком, возвращая ссылки на функции:

var Module = (function () {

var \_privateMethod = function () {

};

var publicMethod = function () {

};

return {

publicMethod: publicMethod,

anotherPublicMethod: anotherPublicMethod

}

})();

6. Основы и особенности javascript. Типы данных. 'use strict'. Всплытия (hoisting). Работа с DOM.

## [Структура кода](https://learn.javascript.ru/javascript-specials" \l "структура-кода)

Операторы разделяются точкой с запятой:

alert('Привет'); alert('Мир');

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

alert('Привет')

alert('Мир')

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

var a = 2

+3

alert(a); // 5

Бывают случаи, когда это ведёт к ошибкам, которые достаточно трудно найти и исправить, например:

alert("После этого сообщения будет ошибка")

[1, 2].forEach(alert)

Детали того, как работает код выше (массивы [...] и forEach) мы скоро изучим, здесь важно то, что при установке точки с запятой после alert он будет работать корректно.

**Поэтому в JavaScript рекомендуется точки с запятой ставить. Сейчас это, фактически, общепринятый стандарт.**

Поддерживаются однострочные комментарии // ... и многострочные /\* ... \*/:

Подробнее: [Структура кода](https://learn.javascript.ru/structure).

## [Переменные и типы](https://learn.javascript.ru/javascript-specials" \l "переменные-и-типы)

* Объявляются директивой var. Могут хранить любое значение:
* var x = 5;

x = "Петя";

* Есть 5 «примитивных» типов и объекты:
* x = 1; // число
* x = "Тест"; // строка, кавычки могут быть одинарные или двойные
* x = true; // булево значение true/false
* x = null; // спец. значение (само себе тип)

x = undefined; // спец. значение (само себе тип)

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

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

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

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

Например:

var age = null; // возраст неизвестен

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

Например:

var x;

alert( x ); // undefined

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

Про объекты мы поговорим в главе [Объекты как ассоциативные массивы](https://learn.javascript.ru/object), они в JavaScript сильно отличаются от большинства других языков.

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

Подробнее: [Переменные](https://learn.javascript.ru/variables), [Шесть типов данных, typeof](https://learn.javascript.ru/types-intro).

## [Строгий режим](https://learn.javascript.ru/javascript-specials" \l "строгий-режим)

Для того, чтобы интерпретатор работал в режиме максимального соответствия современному стандарту, нужно начинать скрипт директивой 'use strict';

'use strict';

...

Эта директива может также указываться в начале функций. При этом функция будет выполняться в режиме соответствия, а на внешний код такая директива не повлияет.

Одно из важных изменений в современном стандарте — все переменные нужно объявлять через var. Есть и другие, которые мы изучим позже, вместе с соответствующими возможностями языка.

## [Взаимодействие с посетителем](https://learn.javascript.ru/javascript-specials" \l "взаимодействие-с-посетителем)

Простейшие функции для взаимодействия с посетителем в браузере:

[**prompt(вопрос[, по\_умолчанию])**](https://developer.mozilla.org/en/DOM/window.prompt)

Задать вопрос и возвратить введённую строку, либо null, если посетитель нажал «Отмена».

[**confirm(вопрос)**](https://developer.mozilla.org/en/DOM/window.confirm)

Задать вопрос и предложить кнопки «Ок», «Отмена». Возвращает, соответственно, true/false.

[**alert(сообщение)**](https://developer.mozilla.org/en/DOM/window.alert)

Вывести сообщение на экран.

Все эти функции являются модальными, т.е. не позволяют посетителю взаимодействовать со страницей до ответа.

Например:

var userName = prompt("Введите имя?", "Василий");

var isTeaWanted = confirm("Вы хотите чаю?");

alert( "Посетитель: " + userName );

alert( "Чай: " + isTeaWanted );

Подробнее: [Взаимодействие с пользователем: alert, prompt, confirm](https://learn.javascript.ru/uibasic).

## [Особенности операторов](https://learn.javascript.ru/javascript-specials" \l "особенности-операторов)

* **Для сложения строк используется оператор +.**

Если хоть один аргумент — строка, то другой тоже приводится к строке:

alert( 1 + 2 ); // 3, число

alert( '1' + 2 ); // '12', строка

alert( 1 + '2' ); // '12', строка

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

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

alert( 0 == false ); // true

alert( true > 0 ); // true

Исключение — сравнение двух строк (см. далее).

**Исключение: значения null и undefined ведут себя в сравнениях не как ноль.**

* + Они равны null == undefined друг другу и не равны ничему ещё. В частности, не равны нулю.
  + В других сравнениях (кроме ===) значение null преобразуется к нулю, а undefined — становится NaN(«ошибка»).

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

Например, забавное следствие этих правил для null:

alert( null > 0 ); // false, т.к. null преобразовано к 0

alert( null >= 0 ); // true, т.к. null преобразовано к 0

alert( null == 0 ); // false, в стандарте явно указано, что null равен лишь undefined

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

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

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

alert( 'а' > 'Я' ); // true

Подробнее: [Основные операторы](https://learn.javascript.ru/operators), [Операторы сравнения и логические значения](https://learn.javascript.ru/comparison).

## [Логические операторы](https://learn.javascript.ru/javascript-specials" \l "логические-операторы)

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

Не стоит путать их с [побитовыми операторами](https://learn.javascript.ru/bitwise-operators) И, ИЛИ, НЕ, которые тоже есть в JavaScript и работают с числами на уровне битов.

Как и в большинстве других языков, в логических операторах используется «короткий цикл» вычислений. Например, вычисление выражения 1 && 0 && 2 остановится после первого И &&, т.к. понятно что результат будет ложным (ноль интерпретируется как false).

**Результатом логического оператора служит последнее значение в коротком цикле вычислений.**

Можно сказать и по-другому: значения хоть и интерпретируются как логические, но то, которое в итоге определяет результат, возвращается без преобразования.

Например:

alert( 0 && 1 ); // 0

alert( 1 && 2 && 3 ); // 3

alert( null || 1 || 2 ); // 1

Подробнее: [Логические операторы](https://learn.javascript.ru/logical-ops).

## [Циклы](https://learn.javascript.ru/javascript-specials" \l "циклы)

* Поддерживаются три вида циклов:
* // 1
* while (условие) {
* ...
* }
* // 2
* do {
* ...
* } while (условие);
* // 3
* for (var i = 0; i < 10; i++) {
* ...

}

* Переменную можно объявлять прямо в цикле, но видна она будет и за его пределами.
* Поддерживаются директивы break/continue для выхода из цикла/перехода на следующую итерацию.

Для выхода одновременно из нескольких уровней цикла можно задать метку.

Синтаксис: «имя\_метки:», ставится она только перед циклами и блоками, например:

outer:

for(;;) {

...

for(;;) {

...

break outer;

}

}

Переход на метку возможен только изнутри цикла, и только на внешний блок по отношению к данному циклу. В произвольное место программы перейти нельзя.

Подробнее: [Циклы while, for](https://learn.javascript.ru/while-for).

## [Конструкция switch](https://learn.javascript.ru/javascript-specials" \l "конструкция-switch)

При сравнениях в конструкции switch используется оператор ===.

Например:

var age = prompt('Ваш возраст', 18);

switch (age) {

case 18:

alert( 'Никогда не сработает' ); // результат prompt - строка, а не число

case "18": // вот так - сработает!

alert( 'Вам 18 лет!' );

break;

default:

alert( 'Любое значение, не совпавшее с case' );

}

Подробнее: [Конструкция switch](https://learn.javascript.ru/switch).

## [Функции](https://learn.javascript.ru/javascript-specials" \l "функции)

Синтаксис функций в JavaScript:

// function имя(список параметров) { тело }

function sum(a, b) {

var result = a + b;

return result;

}

// использование:

alert( sum(1, 2) ); // 3

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

alert( f() ); // undefined

Подробнее: [Функции](https://learn.javascript.ru/function-basics).

## [Function Declaration и Expression](https://learn.javascript.ru/javascript-specials" \l "function-declaration-и-expression)

Функция в JavaScript является обычным значением.

Её можно создать в любом месте кода и присвоить в переменную, вот так:

var sum = function(a, b) {

var result = a + b;

return result;

}

alert( sum(1, 2) ); // 3

Такой синтаксис, при котором функция объявляется в контексте выражения (в данном случае, выражения присваивания), называется Function Expression, а обычный синтаксис, при котором функция объявляется в основном потоке кода — Function Declaration.

Функции, объявленные через Function Declaration, отличаются от Function Expression тем, что интерпретатор создаёт их при входе в область видимости (в начале выполнения скрипта), так что они работают до объявления.

Обычно это удобно, но может быть проблемой, если нужно объявить функцию в зависимости от условия. В этом случае, а также в других ситуациях, когда хочется создать функцию «здесь и сейчас», используют Function Expression.

Детали: [Функциональные выражения](https://learn.javascript.ru/function-declaration-expression).

## [Named Function Expression](https://learn.javascript.ru/javascript-specials" \l "named-function-expression)

Если объявление функции является частью какого-либо выражения, например var f = function... или любого другого, то это Function Expression.

В этом случае функции можно присвоить «внутреннее» имя, указав его после function. Оно будет видно только внутри этой функции и позволяет обратиться к функции изнутри себя. Обычно это используется для рекурсивных вызовов.

Например, создадим функцию для вычисления факториала как Function Expression и дадим ей имя me:

var factorial = function me(n) {

return (n == 1) ? n : n \* me(n - 1);

}

alert( factorial(5) ); // 120

alert( me ); // ошибка, нет такой переменной

Ограничение видимости для имени не работает в IE8-, но вызов с его помощью работает во всех браузерах.

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.

#### **Область видимости в JavaScript**

Одна из причин, приводящих в замешательство новичков, — это область видимости. Вообще, не только новичков. Я встречал много опытных JavaScript-разработчиков, которые не понимают механизм области видимости в JavaScript. Причина в том, что внешне JavaScript очень похож на любой другой Си-подобный язык.  
Давайте рассмотрим следующий код на Cи:

#include <stdio.h>

int main() {

int x = 1;

printf("%d, ", x); // 1

if (1) {

int x = 2;

printf("%d, ", x); // 2

}

printf("%d\n", x); // 1

}

Эта программа выведет 1, 2, 1, потому что Си и все остальные Си-подобные языки реализуют **области видимости на уровне блоков кода**. Когда исполняется новый блок кода, например условие if, новые переменные, объявленные в нём, не повлияют на переменные внешней области видимости.  
Но не в случае JavaScript. Попробуйте запустить вот этот код в Firebug:

var x = 1;

console.log(x); // 1

if (true) {

var x = 2;

console.log(x); // 2

}

console.log(x); // 2

На этот раз будут выведены числа 1, 2, 2. Это связано с тем, что в JavaScript используется **область видимости на уровне функций**. Это совсем не то, что мы привыкли видеть в языках программирования, вроде Си. Блоки кода, вроде того, который у нас идёт сразу после if, **не создают** новую область видимости. Только функции создают новые области видимости.  
Для многих программистов, привыкших к Си, C++, C# или Java такое поведение очень неожиданное и неприятное. К счастью, благодаря гибкости функций JavaScript, можно обойти эту проблему. Чтобы создать временную область видимости внутри функции, достаточно сделать следующее:

function foo() {

var x = 1;

if (x) {

(function () {

var x = 2;

// какой-то код

}());

}

// x всё ещё 1.

}

Такой подход достаточно гибок и может быть использован везде, где вам нужна временная область видимости, не только внутри блоков кода. Но я настаиваю на том, чтобы вы всё-таки потратили своё время, чтобы понять реализацию области видимости в JavaScript. Это довольно мощная особенность языка, которая мне очень нравится. Если вы понимаете область видимости, вам проще будет разобраться в «поднятии» переменных и объявлений функций.

#### **Объявления, именование и «поднятие» переменных и функций**

В JavaScript существует четыре основных способа появления идентификатора в области видимости:

1. **Внутренние механизмы языка**: например, во всех областях видимости доступны this и arguments.
2. **Формальные параметры**: у функций могут быть именованные формальные параметры, область видимости которых ограничена телом функции.
3. **Объявления функций**: объявленные в виде function foo() {}.
4. **Объявления переменных**: например, var foo;.

Интерпретатор JavaScript всегда незаметно для нас перемещает («поднимает») объявления функций и переменных в начало области видимости. Формальные параметры функций и встроенные переменные языка, очевидно, изначально уже находятся в начале. Это значит, что этот код:

function foo() {

bar();

var x = 1;

}

на самом деле интерпретируется так:

function foo() {

var x;

bar();

x = 1;

}

Оказывается, не важно, будет ли вообще выполнена строка, в которой происходит объявление. Следующие две функции эквивалентны:

function foo() {

if (false) {

var x = 1;

}

return;

var y = 1;

}

function foo() {

var x, y;

if (false) {

x = 1;

}

return;

y = 1;

}

Обратите внимание, что присваивание значений переменным не поднимается вместе с их объявлением. Поднимаются только объявления переменных. В случае с функциями, поднимается вся функция целиком. Существуют два основных способа объявить функцию, давайте их рассмотрим:

function test() {

foo(); // TypeError "foo is not a function"

bar(); // "this will run!"

var foo = function () { // функциональное выражение, присвоенное локальной переменной 'foo'

alert("this won't run!");

}

function bar() { // объявление функции с именем 'bar'

alert("this will run!");

}

}

test();

в этом случае поднимается только функция bar. Идентификатор «foo» также поднимается, но не анонимная функция — она остаётся на месте.  
  
Вот мы и описали основные моменты «поднятия» переменных и функций. Конечно, JavaScript не был бы сам собой, если бы не было особых случаев, в которых всё немного сложнее.

#### **Разрешение имён**

Самый важный особый случай, который стоит иметь в виду, — это порядок разрешения имён. Вспомните, есть четыре способа появления идентификаторов в области видимости. Именно в том порядке, в котором я их упомянул, и происходит разрешение имён. В общем случае, если имя уже определено, оно никогда не будет переопределено другой сущностью с таким же именем. То есть объявление функции имеет приоритет над объявлениями переменной с таким же именем. Но это совсем не означает, что присваивание переменной значение не заменит функцию, просто её определение будет проигнорировано.  
Есть несколько исключений:

* Встроенный идентификатор arguments ведёт себя странно. Он как будто объявляется сразу после формальных аргументов функции и перед объявлениями функций. Это означает, что если у функции есть формальный аргумент arguments, у него будет приоритет над встроенным, даже если его не передадут при вызове функции. Это плохая особенность JavaScript. Не используйте формальный аргумент с именем arguments.
* Если вы попробуете использовать this в качестве идентификатора, произойдёт ошибка SyntaxError. Это хорошая особенность.
* Если в списке формальных параметров функции несколько из них имеют одинаковое имя, тот параметр, который упоминается последним, имеет приоритет. Даже если его не передали при вызове функции.

#### **Именованные функциональные выражения**

Вы можете давать имена функциям, определённым с помощью функциональных выражений, используя синтаксис определения функций. Это не приводит к объявлению функции, а следовательно, имя функции ни добавляется в область видимости, ни поднимается вместе с телом функции в начало области видимости. Вот несколько строк, чтобы проиллюстрировать, что я имею в виду:

foo(); // TypeError "foo is not a function"

bar(); // работает

baz(); // TypeError "baz is not a function"

spam(); // ReferenceError "spam is not defined"

var foo = function () {}; // анонимное функциональное выражени (поднимается 'foo')

function bar() {}; // объявление функции (поднимаются 'bar' и тело функции)

var baz = function spam() {}; // именованное функциональное выражение (поднимается только 'baz')

foo(); // работает

bar(); // работает

baz(); // работает

spam(); // ReferenceError "spam is not defined"

Основным инструментом работы и динамических изменений на странице является DOM (Document Object Model) - объектная модель, используемая для XML/HTML-документов.

Согласно DOM-модели, документ является иерархией.  
Каждый HTML-тег образует отдельный элемент-узел, каждый фрагмент текста - текстовый элемент, и т.п.

Проще говоря, DOM - это представление документа в виде дерева тегов. Это дерево образуется за счет вложенной структуры тегов плюс текстовые фрагменты страницы, каждый из которых образует отдельный узел.

[Простейший DOM](http://javascript.ru/tutorial/dom/intro#prosteyshiy-dom)

Построим, для начала, дерево DOM для следующего документа.

|  |  |
| --- | --- |
| 1 | <html> |
| 2 | <head> | |

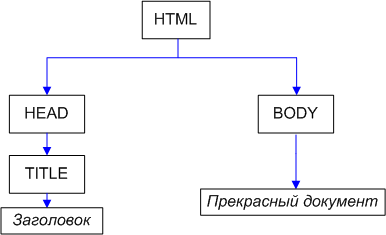
|  |  |  |
| --- | --- | --- |
| 3 | <title>Заголовок</title> | |
| 4 | </head> |

|  |  |
| --- | --- |
| 5 | <body> |
| 6 | Прекрасный документ | |

|  |  |  |
| --- | --- | --- |
| 7 | </body> | |
| 8 | </html> |

Самый внешний тег - <html>, поэтому дерево начинает расти от него.

Внутри <html> находятся два узла: <head> и <body> - они становятся дочерними узлами для <html>.



Теги образуют *узлы-элементы* (element node). Текст представлен *текстовыми узлами*(text node). И то и другое - равноправные узлы дерева DOM.

Очень просто:

Каждый DOM-элемент является объектом и предоставляет свойства для манипуляции своим содержимым, для доступа к родителям и потомкам.

Для манипуляций с DOM используется объект document.  
Используя document, можно получать нужный элемент дерева и менять его содержание.

Например, этот код получает первый элемент с тэгом ol, последовательно удаляет два элемента списка и затем добавляет их в обратном порядке:

[показать чистый исходник в новом окне](http://javascript.ru/tutorial/dom/intro#viewSource)[Скрыть/показать номера строк](http://javascript.ru/tutorial/dom/intro#noList)[печать кода с сохранением подсветки](http://javascript.ru/tutorial/dom/intro#printSource)

|  |  |  |
| --- | --- | --- |
| 1 | var ol = document.getElementsByTagName('ol')[0] | |
| 2 | var hiter = ol.removeChild(ol.firstChild) |

|  |  |  |
| --- | --- | --- |
| 3 | var kovaren = ol.removeChild(ol.firstChild) | |
| 4 | ol.appendChild(kovaren) |

|  |  |
| --- | --- |
| 5 | ol.appendChild(hiter) |

7. Работа с асинхронностью, промисы, стек вызова функций, stack overflow

*Асинхронное программирование* или *Event-driven программирование* - общее название для стиля программирования, при котором результат работы функции приходит не сразу после вызова, а когда-нибудь потом.

В Javascript такое - сплошь и рядом. На этом построен AJAX и javascript-анимация.

Писать программы, в которых результат вызова функции приходит неизвестно когда - куда сложнее, чем обычные. Вложенные вызовы, обработка ошибок, контроль за происходящим - все усложняется.

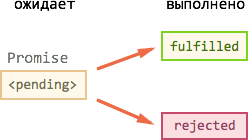
# Promise

Promise (обычно их так и называют «промисы») — предоставляют удобный способ организации асинхронного кода.

В современном JavaScript промисы часто используются в том числе и неявно, при помощи генераторов, но об этом чуть позже.

## [Что такое Promise?](https://learn.javascript.ru/promise" \l "что-такое-promise)

Promise — это специальный объект, который содержит своё состояние. Вначале pending («ожидание»), затем — одно из: fulfilled («выполнено успешно») или rejected («выполнено с ошибкой»).



На promise можно навешивать коллбэки двух типов:

* onFulfilled — срабатывают, когда promise в состоянии «выполнен успешно».
* onRejected — срабатывают, когда promise в состоянии «выполнен с ошибкой».

Способ использования, в общих чертах, такой:

1. Код, которому надо сделать что-то асинхронно, создаёт объект promise и возвращает его.
2. Внешний код, получив promise, навешивает на него обработчики.
3. По завершении процесса асинхронный код переводит promise в состояние fulfilled (с результатом) илиrejected (с ошибкой). При этом автоматически вызываются соответствующие обработчики во внешнем коде.

Синтаксис создания Promise:

var promise = new Promise(function(resolve, reject) {

// Эта функция будет вызвана автоматически

// В ней можно делать любые асинхронные операции,

// А когда они завершатся — нужно вызвать одно из:

// resolve(результат) при успешном выполнении

// reject(ошибка) при ошибке

})

Универсальный метод для навешивания обработчиков:

promise.then(onFulfilled, onRejected)

* onFulfilled — функция, которая будет вызвана с результатом при resolve.
* onRejected — функция, которая будет вызвана с ошибкой при reject.

С его помощью можно назначить как оба обработчика сразу, так и только один:

// onFulfilled сработает при успешном выполнении

promise.then(onFulfilled)

// onRejected сработает при ошибке

promise.then(null, onRejected)

**.catch**

Для того, чтобы поставить обработчик только на ошибку, вместо .then(null, onRejected) можно написать .catch(onRejected) — это то же самое.

**Синхронный throw — то же самое, что reject**

Если в функции промиса происходит синхронный throw (или иная ошибка), то вызывается reject:

'use strict';

let p = new Promise((resolve, reject) => {

// то же что reject(new Error("o\_O"))

throw new Error("o\_O");

})

p.catch(alert); // Error: o\_O

Посмотрим, как это выглядит вместе, на простом примере.

## [Пример с setTimeout](https://learn.javascript.ru/promise" \l "пример-с-settimeout)

Возьмём setTimeout в качестве асинхронной операции, которая должна через некоторое время успешно завершиться с результатом «result»:

'use strict';

// Создаётся объект promise

let promise = new Promise((resolve, reject) => {

setTimeout(() => {

// переведёт промис в состояние fulfilled с результатом "result"

resolve("result");

}, 1000);

});

// promise.then навешивает обработчики на успешный результат или ошибку

promise

.then(

result => {

// первая функция-обработчик - запустится при вызове resolve

alert("Fulfilled: " + result); // result - аргумент resolve

},

error => {

// вторая функция - запустится при вызове reject

alert("Rejected: " + error); // error - аргумент reject

}

);

В результате запуска кода выше — через 1 секунду выведется «Fulfilled: result».

8. Design Patterns. Примеры на js

Decorator

Чертовски приятный паттерн, с его помощью можно менять поведение объекта на лету, в зависимости от каких-нибудь условий.   
  
Допустим, у нас есть такой код:

function Ball( param )

{

this.\_radius = param.radius;

this.\_color = param.color;

}

Ball.prototype =

{

constructor: Ball,

INCREMENTATION\_STEP: 5,

draw: function(){console.log("ball drawn with radius:" + this.\_radius + " and color: " + this.\_color)},

inc: function(){ this.\_radius += this.INCREMENTATION\_STEP }

}

new Ball({ radius:100, color:"red"});

Здесь мы создаем новый красный мячик, а что делать если мячик нужен не просто красный, а красный в полоску? Вот тут на сцену и выходит Decorator.  
  
Особый шарм ему придает то, что первоначальный *Ball* вообще не подозревает о том, что он может быть в полоску, или что у него могут быть какие-то там декораторы.   
  
Реализовать паттерн можно несколькими способами:

Способ первый — комплексный

function StripedBall( ball )

{

this.\_ball = ball

}

StripedBall.prototype =

{

constructor: StripedBall,

draw: function()

{

this.\_ball.draw();

console.log("and with stripes");

},

inc: function()

{

return this.\_ball.inc();

}

}

function SpeckledBall( ball )

{

this.\_ball = ball

}

SpeckledBall.prototype =

{

constructor: SpeckledBall,

draw: function()

{

this.\_ball.draw();

console.log("and with dots!");

},

inc: function()

{

return this.\_ball.inc();

}

}

В каждом декораторе нужно воссоздать все функции которые должны быть в объекте родителе, и в тех из них, поведение которых мы менять не хотим, нужно просто перенаправлять запрос родителю. Этот способ лучше применять когда происходят серьезные изменения, которые затрагивают > 1 — 2 функций  
  
Пишем простенький тест:

var ball1 = new SpeckledBall( new StripedBall( new Ball({ radius:100, color:"red"})));

var ball2 = new StripedBall( new SpeckledBall( new Ball({ radius:100, color:"green"})));

ball1.draw();

ball1.inc();

ball1.draw();

ball2.draw();

Глубокий вздох, и проверка:

ball drawn with radius:100 and color: red

and with stripes

and with dots!

ball drawn with radius:105 and color: red

and with stripes

and with dots!

ball drawn with radius:100 and color: green

and with dots!

and with stripes

Зря волновался — работает все как надо.

Способ второй — легковесный

function MakeStripedBall( ball )

{

var function\_name = "draw";

var prev\_func = ball[ function\_name ];

ball[ function\_name ] = function()

{

prev\_func.apply( this, arguments )

console.log("and with stripes");

};

return ball;

}

function MakeSpeckledBall( ball )

{

var function\_name = "draw";

var prev\_func = ball[function\_name];

ball[function\_name] = function ()

{

prev\_func.apply(this, arguments)

console.log("and with dots!");

};

return ball;

}

Кода, конечно, нужно меньше чем в первом случае, зато, если изменяемых функций больше чем 1-2, или изменения комплексные — разобраться во всем этом будет намного сложнее.  
  
Пишем тест:

var ball3 = MakeStripedBall( MakeSpeckledBall( new Ball({ radius: 150, color: "blue" })));

var ball4 = MakeSpeckledBall( MakeStripedBall(new Ball({ radius: 150, color: "blue" })));

ball3.draw();

ball3.inc();

ball3.draw();

ball4.draw();

И проверяем, как все это работает:

ball drawn with radius:150 and color: blue

and with dots!

and with stripes

ball drawn with radius:155 and color: blue

and with dots!

and with stripes

ball drawn with radius:150 and color: blue

and with stripes

and with dots!

Все как надо.

Factory

Собственно, основной задачей фабрики в статически типизируемых языках является создание разных объектов с одинаковым интерфейсом, в зависимости от ситуаций, в JavaScript этак проблема так остро не стоит, так что появляется вопрос — зачем эта фабрика тут вообще нужна?   
  
Все просто — помимо этой, первой, цели, у нее есть еще и вторая — фабрика может проводить какую-то первичную инициализацию объектов.  
  
Например, предположим, у нас есть объекты Daddy, Mammy, и lad, создавая их с помощью фабрики мы можем просто сказать — *familyfactory.createLad(); familyfactory.createDaddy()*, а уж то, что они оба рыжие и 210см. роста, за нас решит фабрика — эти параметры мы не задаем.  
  
Собственно, для того чтобы фабрика могла создавать какие-то объекты, для них сначала неплохо бы задать конструкторы (в этом примере объекты, к сожалению, не такие интересные как несколькими строками выше ):

var Shapes =

{

Circle: function (param)

{

console.log("new " + param.color + " circle created with radius " + param.radius + "px");

},

Square: function (param)

{

console.log("new " + param.color + " square created with " + param.side + "px on a side ");

},

Triangle: function (param)

{

console.log("new " + param.color + " triangle created with " + param.side + "px on a side ");

}

}

А теперь можно сделать и саму фабрику — выглядеть она может так:

function ShapeFactory(size, color)

{

this.size = size;

this.color = color;

}

ShapeFactory.prototype =

{

constructor: ShapeFactory,

makeCircle: function () { return new Shapes.Circle({ radius: this.size / 2, color: this.color }); },

makeSquare: function () { return new Shapes.Square({ side: this.size, color: this.color }); },

makeTrinagle: function () { return new Shapes.Triangle({ side: this.size, color: this.color }); }

}

Пишем скромненький тест:

var factory = new ShapeFactory(100, "red")

factory.makeSquare();

factory.makeSquare();

factory.makeTrinagle();

factory.makeCircle();

factory.makeTrinagle();

И смотрим в консоль:

new red square created with 100px on a side

new red square created with 100px on a side

new red triangle created with 100px on a side

new red circle created with radius 50px

new red triangle created with 100px on a side

Всё работает

Singleton

Что же такое синглтон? Объяснение будет сложным, долгим и нетривиальным — это объект, который есть в системе в одном экземпляре. Тадаам — конец объяснения.  
  
Я частенько вообще не задумываюсь над тем, что это вообще-то тоже паттерн. Стоит сказать что перед тем как его применять стоит хорошенько подумать — на самом деле нужен синглетон не очень часто.   
Сделать его можно несколькими способами.

Способ первый — тривиальный

var singleton\_A =

{

log: function( text ){ console.log(text); }

}

Это простой наглядный и эффективный метод, который, даже, в объяснении, по-моему, не нуждается.

Способ второй — выпендрежный

Основная его задача — это показать какой ты крутой однокурсникам или другим джуниорам. Кроме этого он, конечно, может быть действительно полезен — с таким подходом проще перестраиваться если планы изменились и где-то в середине проекта синглетон решили заменить несколькими объектами

var Singleton\_B;

(function(){

var instance;

var anticlone\_proxy;

Singleton\_B = function(){

if( instance ){ return instance; }

instance =

{

\_counter: 0,

log: function( text ){ this.\_counter++; console.log( text + this.\_counter ); }

}

anticlone\_proxy =

{

log: function( text ){ return instance.log( text ); }

}

return anticlone\_proxy;

};

})();

Его фишка в том что мы просто создаем объект, а синглетон он, или нет — нас в общем-то не очень волнует:

function NonSingleton() { }

NonSingleton.prototype =

{

consturctor: NonSingleton,

scream: function(){console.log("Woooohoooooo!")}

}

var singleton = new Singleton\_B();

var nonsingleton = new NonSingleton();

singleton.log("3..2..1... ignition!");

nonsingleton.scream();

Если этот код выполнить, то в консоли мы увидим:

3..2..1... ignition!

Woooohoooooo!

Memoization

Очень простая и полезная техника — суть её в том, что для функции которая может долго вычислять результат, мы создаем небольшой кэш ответов. Работает это, разумеется, только в том случае, когда при одинаковых входных параметрах результат функции тоже должен быть одинаковый.  
  
Создаем какую-нибудь медленную функцию, которая использует эту технику:

function calculation(x, y)

{

var key = x.toString() + "|" + y.toString();

var result = 0;

if (!calculation.memento[key])

{

for (var i = 0; i < y; ++i) result += x;

calculation.memento[key] = result;

}

return calculation.memento[key];

}

calculation.memento = {};

И проверяем сколько мы можем выйграть времени:

console.profile();

console.log('result:' + calculation(2, 100000000));

console.profileEnd();

console.profile();

console.log('result:' + calculation(2, 100000000));

console.profileEnd();

console.profile();

console.log('result:' + calculation(2, 10000000));

console.profileEnd();

Если этот код теперь запустить в FF с Firebug, то мы увидим следующую статистику:

Profile1: 626.739ms

result:200000000

0.012ms

result:200000000

63.055msresult:20000000

Как видно из логов — при повторном запросе мы сэкономили кучу времени.

Mediator

Mediator — это такая штука, которая помогает в особо запущенных случаях взаимодействия между обьектами, например, когда у нас, скажем, 5 обьектов более-менее разного типа, и все почему-то знают друг о друге, стоит серьезно задуматься о медиаторе.  
  
В качестве подготовки сначала сделаем несколько классов, которые в перспективе медиатор будут использовать (подсказка в данном случае медиатор будет называтья kitchen:)

function Daddy() { }

Daddy.prototype =

{

constructor: Daddy,

getBeer: function ()

{

if (!kitchen.tryToGetBeer())

{

console.log("Daddy: Who the hell drank all my beer?");

return false;

}

console.log("Daddy: Yeeah! My beer!");

kitchen.oneBeerHasGone();

return true;

},

argue\_back: function () { console.log("Daddy: it's my last beer, for shure!"); }

}

function Mammy() { }

Mammy.prototype =

{

constructor: Mammy,

argue: function ()

{

console.log("Mammy: You are f\*king alconaut!");

kitchen.disputeStarted();

}

}

function BeerStorage(beer\_bottle\_count)

{

this.\_beer\_bottle\_count = beer\_bottle\_count;

}

BeerStorage.prototype =

{

constructor: BeerStorage,

takeOneBeerAway: function ()

{

if (this.\_beer\_bottle\_count == 0) return false;

this.\_beer\_bottle\_count--;

return true;

}

}

А теперь пора написать и сам медиатор:

var kitchen =

{

daddy: new Daddy(),

mammy: new Mammy(),

refrigerator: new BeerStorage(3),

stash: new BeerStorage(2),

tryToGetBeer: function ()

{

if (this.refrigerator.takeOneBeerAway()) return true;

if (this.stash.takeOneBeerAway()) return true;

return false

},

oneBeerHasGone: function (){ this.mammy.argue(); },

disputeStarted: function (){ this.daddy.argue\_back(); }

}

И так, у нас есть 4 объекта работа со взаимодействием между которыми, могла бы превратиться в неплохое наказание, если бы проходила не через Mediator.  
  
Пишем код проверки:

var round\_counter = 0;

while (kitchen.daddy.getBeer())

{

round\_counter++

console.log( round\_counter + " round passed");

}

Спрашиваем у консоли — все ли идет по плану:

Daddy: Yeeah! My beer!

Mammy: You are f\*king alconaut!

Daddy: it's my last beer, for shure!

1 round passed

...

Daddy: Yeeah! My beer!

Mammy: You are f\*king alconaut!

Daddy: it's my last beer, for shure!

5 round passed

Daddy: Who the hell drank all my beer?

Некоторую часть этой душевной беседы я вырезал, но в целом все как надо.

Observer

Это тот самый паттерн, который мы используем по пятьдесят раз в день даже особенно об этом незадумывасяь — $(#some\_useful\_button).click( blah\_blah\_blah ) — знакомая конструкция? В ней click — это событие, а blah\_blah\_blah какраз и есть тот самый Observer который за этим событием наблюдает.  
  
Для своих коварных планов, можно реализовать и собственную систему событий и наблюдателей, которая будет уметь реагировать не только на действия пользвоателя, но и на что-нибудь еще.  
  
Ключевым её компонентом является объект событие:

Event = function()

{

this.\_observers = [];

}

Event.prototype =

{

raise: function (data)

{

for (var i in this.\_observers)

{

var item = this.\_observers[i];

item.observer.call(item.context, data);

}

},

subscribe: function (observer, context)

{

var ctx = context || null;

this.\_observers.push({ observer: observer, context: ctx });

},

unsubscribe: function (observer, context )

{

for (var i in this.\_observers)

if ( this.\_observers[i].observer == observer &&

this.\_observers[i].context == context )

delete this.\_observers[i];

}

}

Вообще, я тут подумал, что как-то скучно без скриншотов и ссылок, поэтому в этот раз примера будет два.

Первый — простой

var someEvent = new Event();

someEvent.subscribe(function ( data ) { console.log("wohoooooo " + data ) });

var someObject =

{

\_topSecretInfo: 42,

observerFunction: function () { console.log("Top Secret:" + this.\_topSecretInfo) }

}

someEvent.subscribe(someObject.observerFunction, someObject);

someEvent.raise("yeaah!");

someEvent.raise();

И консоль подтверждает что все работает.

wohoooooo yeaah!

Top Secret:42

wohoooooo undefined

Top Secret:42

9. ES6

Так как изменений очень много, опишу субъективно самые важные. 

#### **let и scope — переменные с блочной областью видимости (block scope)**

Одна из самых раздражающих/непонятных для начинающих JS-программистов вещей это область видимости переменных в JS. Пример:

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

console.log(i);

В большинстве языков подобный код выкинет ошибку что i не дефинирована, но в JS в консоль выведется число «10». Причина в том, что в JS используется [hoisting](http://www.w3schools.com/js/js_hoisting.asp) — то есть декларации всех используемых переменных переносятся в начало функции (и соответственно i доступна вне блока for). Именно поэтому советуют декларировать все переменные в начале функции — все равно они будут туда перенесены при выполнении кода.  
В ES6 можно написать:

for(let j=0; j<10; j++){ }

console.log(j);

В этом коде область видимости **j** ограничена блоком for. Больше нет смысла декларировать переменные в начале функции!  
  
Про **const** говорить особо нечего — это декларация константы, при повторном присвоении значения выбрасывает TypeError (в strict mode).  
  
Пример:

function setConst(){

"use strict";

const xx = 10;

xx = 11;

console.log(xx);

}

setConst();

Ссылок на примеры нет, так как на данный момент block scope в es6fiddle не поддерживается. Ждем фиксов.

#### **Arrow functions**

Если вы использовали лямбды в C#, вы сразу узнаете синтаксис (это не случайность — он был [взят именно оттуда](http://wiki.ecmascript.org/doku.php?id=harmony:arrow_function_syntax)).   
Как писали раньше:

function SomeClass() {

var self = this; //самый простой способ сохранить контекст нужного объекта

self.iter = 0;

setInterval(function() {

self.iter++;

console.log('current iteration: ' + self.iter);

}, 1000);

}

var sc = new SomeClass();

В ES6 можно гораздо проще:

function SomeClass() {

this.iter = 0;

setInterval(() => {

this.iter++;

console.log('current iteration: ' + this.iter);

}, 1000);

}

var sc = new SomeClass();

[es6fiddle](http://www.es6fiddle.net/i2aadppx/)  
  
Как видно, arrow function не создает своего контекста, так что this будет указывать на контекст функции, в которой вызван код.  
  
Пример с параметрами:

let numList = [1,2,3];

let doubleNumList = numList.map(n => n\*2);

console.log(doubleNumList);

[es6fiddle](http://www.es6fiddle.net/i2aba5e4/)  
  
Сравните, насколько более громоздкий старый метод:

let numList = [1,2,3];

let doubleNumList = numList.map(function(n){ return n\*2; });

console.log(doubleNumList);

#### **Классы**

Старый способ эмулировать классы в JS выглядит как-то так:

function Vehicle(topSpeed){

this.topSpeed = topSpeed;

this.printTopSpeed = function(){

console.log('Top speed:'+this.topSpeed+' km/h');

}

}

var myVehicle = new Vehicle(50);

myVehicle.printTopSpeed();

[jsFiddle](http://jsfiddle.net/dvg0vr32/)  
  
«Традиционное» наследование как в Java/C# можно также эмулировать через протопипную модель (код приводить не буду, чтобы не раздувать статью).   
В ES6 появляются «настоящие» классы:

class Vehicle {

constructor(topSpeed){

this.topSpeed = topSpeed;

}

printTopSpeed(){

console.log('Top speed:'+this.topSpeed+' km/h');

}

}

class Bicycle extends Vehicle {

constructor(topSpeed, wheelSize, bicycleType, producedBy){

super(topSpeed);

this.wheelSize = wheelSize;

this.bicycleType = bicycleType;

this.producedBy = producedBy;

}

static wheelCount(){ return 2; }

get bikeInfo(){

return this.producedBy + ' ' + this.bicycleType + ' bike';

}

printBicycleType(){

console.log('Type:'+this.bicycleType+' bike');

}

}

var myBike = new Bicycle(40,622,'road','Trek');

myBike.printTopSpeed();

myBike.printBicycleType();

console.log('Bicycles have '+Bicycle.wheelCount()+' wheels');

console.log(myBike.bikeInfo);

[es6fiddle](http://www.es6fiddle.net/i2ae1wt4/)  
  
Думаю, что код выше в комментариях не нуждается. Как видно, в классах можно дефинировать геттеры и статические методы (но не поля), синтаксис интуитивно понятен. Стоит отметить, что тело класса (class body) всегда интерпретируется в strict mode. Ну и конечно никто не заставляет оголтелых адептов прототипного наследования менять свои привычки — классы это вообщем-то синтаксический сахар поверх старой модели.

#### **Destructuring assignment**

Напоследок, небольшая но интересная фича, которая облегчает присвоение значений. Пример:

function getFirstPrimeNumbers(){

return [2,3,5];

}

var [firstPrime, secondPrime, thirdPrime] = getFirstPrimeNumbers();

console.log(thirdPrime); //5

//обмен значений (value swap)

var [x,y] = [1,2];

console.log('x:'+x, 'y:'+y); //x:1 y:2

[x,y] = [y,x];

console.log('x:'+x, 'y:'+y); //x:2 y:1

[es6fiddle](http://www.es6fiddle.net/i2amsvuv/)  
  
Теперь можно обменивать значения без временной переменной, что сокращает код.  
  
Немного более сложный пример:

var myBook = {

title: "Surely You're Joking, Mr. Feynman!",

author:{

firstName: 'Richard',

lastName: 'Feynman',

yearBorn: 1918

}

};

function getTitleAndAuthorsLastName({ title, author: { lastName } }){

return 'title: '+ title + ' last name:'+lastName;

}

console.log(getTitleAndAuthorsLastName(myBook));

10. События, обработка событий, Event Bubbling, Event Capturing.

Практически все JavaScript-приложения выполняют те или иные действия, откликаясь на различные события.

Событие - это сигнал от браузера о том, что что-то произошло.

Есть множество самых различных событий.

* DOM-события, которые инициируются элементами DOM. Например, событиеclick происходит при клике на элементе, а событие mouseover - когда указатель мыши появляется над элементом,
* События окна. Например событие resize - при изменении размера окна браузера,
* Другие события, например load, readystatechange. Они используются, скажем, в технологии AJAX.

Именно DOM-события связывают действия, происходящие в документе, с кодом JavaScript, тем самым обеспечивая динамический веб-интерфейс.

## [Назначение обработчиков](http://javascript.ru/tutorial/events/intro#naznachenie-obrabotchikov)

Для того, чтобы скрипт реагировал на событие - нужно назначить хотя бы одну функцию-обработчик. Обычно обработчики называют "on+имя события", например:onclick.

Нужно сразу отметить, что JavaScript - однопоточный язык, поэтому обработчики всегда выпоняются последовательно и в общем потоке. Это значит, что при установке обработчиков двух событий, которые возникают на элементе одновременно, напримерmouseover (мышь появилась над элементом) и mousemove (мышь двигается над элементом), их обработчики будут выполнены последовательно.

Существует несколько способов назначать обработчик на конкретное событие элемента. Все они представлены ниже.

### [Через атрибут HTML-тега](http://javascript.ru/tutorial/events/intro#cherez-atribut-html-tega)

Обработчик события можно указать в виде inline-записи, прямо в атрибутеon*событие*.

Например, для обработки события click на кнопке input, можно назначить обработчик onclick вот так:

|  |
| --- |
| <input id="b1" value="Нажми Меня" onclick="alert('Спасибо!');" type="button"/> |

Этот код в действии:

Можно назначить и функцию.

Например, пусть при клике на кнопку input запускается функция count\_rabbits(). Для этого запишем вызов функции в атрибут onclick:

|  |  |  |
| --- | --- | --- |
| 01 | <html> | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | <head> |
| 04 | <script type="text/javascript"> | |

|  |  |
| --- | --- |
| 05 | function count\_rabbits() { |
| 06 | for(var i=1; i<=3; i++) { | |

|  |  |
| --- | --- |
| 07 | // оператор + соединяет строки |
| 08 | alert("Из шляпы достали "+i+" кролика!") | |

|  |  |  |
| --- | --- | --- |
| 09 | } | |
| 10 | } |

|  |  |  |
| --- | --- | --- |
| 11 | </script> | |
| 12 | </head> |

|  |  |
| --- | --- |
| 13 |  |
| 14 | <body> | |

|  |  |  |
| --- | --- | --- |
| 15 | **<input type="button" onclick="count\_rabbits()" value="Считать кролей!"/>** | |
| 16 | </body> |

|  |  |
| --- | --- |
| 17 |  |
| 18 | </html> | |

Напомним, что имена атрибутов HTML-тегов нечувствительны к регистру, поэтому атрибут oNcLiCk сработает также, как onClick или onclick.

Но если вы хотите придерживаться хорошего стиля (или спецификации XHTML), то имена тегов и их атрибуты должны быть указаны в нижнем регистре.

#### [**Когда использовать**](http://javascript.ru/tutorial/events/intro#kogda-ispolzovat)

Такой способ установки обработчиков очень удобен - он нагляден и прост, поэтому часто используется в решении простых задач.

У этого способа установки обработчика есть и минусы. Как только обработчик начинает занимать больше одной строки - читабельность резко падает.

Впрочем, сколько-нибудь сложные обработчики в HTML никто не пишет. Вместо этого лучше устанавливать обработчики из JavaScript способами, которые будут представлены ниже.

* Просто для простых задач
* Смесь javascript-кода и HTML-разметки
* Сложные обработчики писать сложно или невозможно

### [Через свойство объекта](http://javascript.ru/tutorial/events/intro#cherez-svoystvo-obekta)

Самый близкий родственник описанного выше способа - установка функции-обработчика через свойство on*событие* соответствующего элемента. Этот способ тоже будет работать в любом браузере с поддержкой JavaScript.

Для этого нужно:

1. получить элемент
2. назначить обработчик свойству on+имя

Вот пример установки обработчика события click на элемент с id="myElement":

|  |  |
| --- | --- |
| document.getElementById('myElement').onclick = function() { | |
| alert('Спасибо') |

|  |
| --- |
| } |
| <input id="myElement" type="button" value="Нажми меня"/> | |

Этот код в действии:

Стоит сразу обратить внимание на две детали:

1. Это именно свойство, а не атрибут. Поэтому, хотя технически и есть кроссбраузерные способы назначать обработчики через setAttribute, но лучше их даже не знать, а пользоваться прямым присвоением.

Кроме того, как и все свойства объектов JavaScript, имя свойства on*событие*чувствительно к регистру символов и должно быть в нижнем регистре.

1. Обработчик - не текст, а именно функция javascript.

Когда браузер видит свойство on... в HTML-разметке - он создает функцию из содержимого кавычек.

В этом смысле эти два кода работают одинаково:

1. Только HTML:

|  |
| --- |
| <input type="button" onclick=" alert('Клик!') "/> |

1. HTML + JS:

|  |
| --- |
| <input type="button" id="button"/> |
| document.getElementById('button').onclick = function() { | |

|  |  |
| --- | --- |
| alert('Клик') | |
| } |

Все вызовы типа getElementById должны запускаться после описания соответствующего HTML-узла, а лучше - после окончания загрузки страницы.

Иначе узел просто не будет найден.

Конечно, можно и не создавать анонимную функцию, а использовать любую уже готовую:

|  |  |  |
| --- | --- | --- |
| 1 | function doSomething() { | |
| 2 | alert('Спасибо') |

|  |  |  |
| --- | --- | --- |
| 3 | } | |
| 4 |  |

|  |  |
| --- | --- |
| 5 | document.getElementById('button').onclick = doSomething |

*Частая ошибка новичков*

Обратите внимание - свойству присваивается именно сама функция-обработчик doSomething, а не doSomething():

|  |
| --- |
| document.getElementById('button').onclick = doSomething |

doSomething() - это результат запуска функции, а так как вызоваreturn в ее коде нет, то этот результат будет undefined.

Сравните это с атрибутом. Там - наоборот, скобки нужны:

|  |
| --- |
| <input type="button" id="mybutton" onclick="doSomething()"/> |

Это различие легко объяснить. Дело в том, что при назначенииonclick в HTML браузер автоматически создает функцию-обработчик из содержимого кавычек. Получается, что последний пример - это по сути то же самое, что:

|  |  |
| --- | --- |
| document.getElementById('mybutton').onclick = function() { | |
| doSomething()  // внутри автосозданной функции |

|  |
| --- |
| } |

#### [**Когда использовать**](http://javascript.ru/tutorial/events/intro#kogda-ispolzovat-1)

Описанная установка обработчика через свойство - очень популярный и простой способ.

У него есть один недостаток: на элемент можно повесить только один обработчик нужного события.

Например:

|  |  |
| --- | --- |
| input.onclick = function() { alert(1) } | |
| // ... |

|  |
| --- |
| input.onclick = function() { alert(2) }  // заменит предыдущий |

1. Удобный и надежный способ, работает из javascript
2. Только один обработчик на событие

Конечно, можно при назначении нового обработчика копировать предыдущий и запускать его самостоятельно. Но лучше использовать специальные методы назначения.

### [Специальные методы](http://javascript.ru/tutorial/events/intro#specialnye-metody)

Представленных выше методов недостаточно для случаев, которые возникают при разработке серьёзного JavaScript-приложения.

Классический пример - установка обработчика на событие "содержимое окна загрузилось":

|  |  |
| --- | --- |
| 1 | // разные элементы интерфейса могут иметь интерес |
| 2 | // в том, чтобы их вызвали при загрузке документа |

|  |  |
| --- | --- |
| 3 | window.onload = function() { |
| 4 | alert('Документ загружен!') | |

|  |  |
| --- | --- |
| 5 | } |

Если заведомо нет локальной переменной onload, то можно и не упоминать про window. Пустячок, но код немного короче.

|  |
| --- |
| onload = function() { ... } |

Существует два основных интерфейса для установки событий.

#### [**Решение Microsoft**](http://javascript.ru/tutorial/events/intro#reshenie-microsoft)

Методы, предложенные Microsoft, работают только в браузерах Internet Explorer и Opera(она поддерживает метод Microsoft для лучшей совместимости).

Установка обработчика:

|  |
| --- |
| element.attachEvent( "on"+имя события, обработчик) |

Удаление обработчика:

|  |
| --- |
| element.detachEvent( "on"+имя события, обработчик) |

Например:

|  |  |  |
| --- | --- | --- |
| 1 | var input = document.getElementById('b1') | |
| 2 | var handler = function() { |

|  |  |  |
| --- | --- | --- |
| 3 | alert('Спасибо!') | |
| 4 | } |

|  |  |  |
| --- | --- | --- |
| 5 | input.attachEvent( "onclick" , handler) // поставить обработчик | |
| 6 | // .... |

|  |  |
| --- | --- |
| 7 | input.detachEvent( "onclick", handler) // убрать обработчик |

*Частая ошибка новичков*

Обратите внимание - установка и удаление обработчика оперируют одной и той же функцией handler.

Так было бы неправильно:

|  |  |
| --- | --- |
| 1 | input.attachEvent( "onclick" , |
| 2 | function() {alert('Спасибо')} | |

|  |  |
| --- | --- |
| 3 | ) |
| 4 | // .... | |

|  |  |
| --- | --- |
| 5 | input.detachEvent( "onclick", |
| 6 | function() {alert('Спасибо')} | |

|  |  |
| --- | --- |
| 7 | ) |
| 8 | // function(){} создает две разные функции | |

Поэтому при назначении обработчика, если вы планируете его потом снимать - необходимо где-то сохранить указатель на функцию.

Как уже говорилось ранее, вы можете установить несколько обработчиков на одно событие одного элемента:

|  |  |  |
| --- | --- | --- |
| 01 | var myElement2 = document.getElementById("myElement2"); | |
| 02 | var handler = function() { |

|  |  |  |
| --- | --- | --- |
| 03 | alert('Спасибо!'); | |
| 04 | } |

|  |  |
| --- | --- |
| 05 |  |
| 06 | var handler2 = function() { | |

|  |  |  |
| --- | --- | --- |
| 07 | alert('Еще раз спасибо!'); | |
| 08 | } |

|  |  |
| --- | --- |
| 09 |  |
| 10 | myElement2.attachEvent("onclick", handler); | |

|  |  |
| --- | --- |
| 11 | myElement2.attachEvent("onclick", handler2); |

Этот код в действии (будет работать только в Internet Explorer/Opera):

#### [**Установка по стандарту W3C**](http://javascript.ru/tutorial/events/intro#ustanovka-po-standartu-w3c)

Решение W3C работает во всех современных браузерах, кроме Internet Explorer.

Установка обработчика:

|  |
| --- |
| element.addEventListener( имя\_события, обработчик, фаза) |

Удаление обработчика:

|  |
| --- |
| element.removeEventListener( имя\_события, обработчик, фаза) |

Обратите внимание, что имя события указывается без префикса "on".

Еще одно отличие от решения Microsoft это третий параметр – фаза.  
Если он установлен в true, то при срабатывании события во вложенном элементе, обработчик будет вызван на фазе "перехвата", а если значение будет false, то - на фазе "всплывания". Подробнее об этом будет написано далее, в разделе этой статьи "Порядок срабатывания событий".

При обычной установке обработчика третий параметр всегда должен быть false.

Использование - аналогично решению от Microsoft:

|  |  |
| --- | --- |
| 1 | // ... объявить функцию-обработчик handler ... |
| 2 | input.addEventListener( "click" , handler, false) // поставить обработчик | |

|  |  |
| --- | --- |
| 3 | // .... |
| 4 | input.removeEventListener( "click", handler, false) // убрать обработчик | |

Как и в других случаях, вы должны передать имя обработчика не ставя круглых скобок, иначе функция будет выполнена сразу, а в качестве обработчика будет передан лишь её результат.

1. Сколько угодно обработчиков
2. Кросс-браузерные несовместимости

Далее мы вернемся к различным способам установки обработчиков и в подробностях рассмотрим, как сделать все кросс-браузерно.

## [Объект "событие" (event)](http://javascript.ru/tutorial/events/intro#obekt-sobytie-event)

Объект событие всегда передается обработчику и содержит массу полезной информации о том где и какое событие произошло.

Способов передачи этого объекта обработчику существует ровно два, и они зависят от способа его установки и от браузера.

### [W3C](http://javascript.ru/tutorial/events/intro#w3c)

В браузерах, работающих по рекомендациям W3C, объект события всегда передается в обработчик первым параметром.

Например:

|  |  |
| --- | --- |
| 1 | function doSomething(event) { |
| 2 | // event - будет содержать объект события | |

|  |  |  |
| --- | --- | --- |
| 3 | } | |
| 4 |  |

|  |  |
| --- | --- |
| 5 | element.onclick = doSomething; |

При вызове обработчика объект события event будет передан ему первым аргументом.

Можно назначить и вот так:

|  |  |
| --- | --- |
| element.onclick = function(event) { | |
| // event - объект события |

|  |
| --- |
| } |

Интересный побочный эффект - в возможности использования переменной event при назначении обработчика в HTML:

|  |
| --- |
| <input type="button" onclick="alert(event)" value="Жми сюда не ошибешься"/> |

Это работает благодаря тому, что браузер автоматически создает функцию-обработчик с данным телом, в которой первый аргумент event.

### [Internet Explorer](http://javascript.ru/tutorial/events/intro#internet-explorer)

В Internet Explorer существует глобальный объект window.event, который хранит в себе информацию о последнем событии. А первого аргумента обработчика просто нет.

То есть, все должно работать так:

|  |  |  |
| --- | --- | --- |
| 1 | // обработчик без аргументов | |
| 2 | function doSomething() { |

|  |  |  |
| --- | --- | --- |
| 3 | // window.event - объект события | |
| 4 | } |

|  |  |
| --- | --- |
| 5 |  |
| 6 | element.onclick = doSomething; | |

Обратите внимание, что доступ к event при назначении обработчика в HTML (см. пример выше) по-прежнему будет работать. Такой вот надежный и простой кросс-браузерный доступ к объекту события.

### [Кросс-браузерное решение](http://javascript.ru/tutorial/events/intro#kross-brauzernoe-reshenie)

Можно кросс-браузерно получить объект события, использовав такой приём:

|  |  |
| --- | --- |
| 1 | function doSomething(event) { |
| 2 | event = event || window.event | |

|  |  |
| --- | --- |
| 3 |  |
| 4 | // Теперь event - объект события во всех браузерах. | |

|  |  |  |
| --- | --- | --- |
| 5 | } | |
| 6 |  |

|  |  |
| --- | --- |
| 7 | element.onclick = doSomething |

### [Получение события при inline-записи](http://javascript.ru/tutorial/events/intro#poluchenie-sobytiya-pri-inline-zapisi)

Как мы уже говорили раньше, при описании обработчика события в HTML-разметке для получения события можно использовать переменную с названием event.

|  |
| --- |
| <input type="button" onclick="alert(event.type)" value="Нажми меня"/> |

Этот код в действии:

Это совершенно кросс-браузерный способ, так как по стандарту event - название первого аргумента функции-обработчика, которую автоматом создаст браузер; ну а в IE значение event будет взято из глобального объекта window.

## [Что дает объект события?](http://javascript.ru/tutorial/events/intro#chto-daet-obekt-sobytiya)

Из объекта события обработчик может узнать, на каком элементе оно произошло, каковы были координаты мыши (для событий, связанных с мышью), какая клавиша была нажата (для событий, связанных с клавиатурой), и извлечь другую полезную информацию.

Например, для события по клику мыши (onclick), свойство event.target(в IEevent.srcElement) содержит DOM-элемент, на котором этот клик произошел.

Более подробно это описано в следующей статье [Свойства объекта событие](http://javascript.ru/tutorial/events/properties).

## [Порядок срабатывания событий](http://javascript.ru/tutorial/events/intro#poryadok-srabatyvaniya-sobytiy)

Примечательно, что на одно событие может реагировать не только тот элемент, на котором произошло событие, но и элементы над ним.

Это очень удобно, например если в элементе содержатся много дочерних HTML-тегов - не обязательно ставить обработчик на каждый, достаточно указать один обработчик на родителе и в нем ловить все события.

Рассмотрим ситуацию, когда у вас есть три элемента "вложенных" друг в друга.

1

2

3

|  |  |  |
| --- | --- | --- |
| 1 | <div class="d1" >1<!-- самый верхний, в представлении DOM, элемент --> | |
| 2 | <div class="d2">2 |

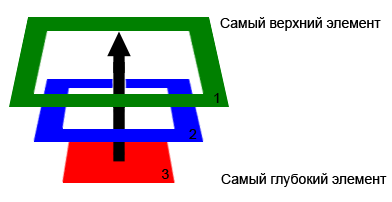
|  |  |  |
| --- | --- | --- |
| 3 | <div class="d3">3</div><!-- самый глубокий элемент --> | |
| 4 | </div> |

|  |  |
| --- | --- |
| 5 | </div> |

Если на каждом из них будет свой обработчик события, например onclick, то обработчик для какого элемента будет вызван первым при клике, скажем, на d3?

Всего существует 2 модели поведения, они не имеют преимуществ между собой, но используют принципиально разные подходы. Стандарт W3C объединяет две модели в одну универсальную.

### [Всплывающие события (Bubbling)](http://javascript.ru/tutorial/events/intro#vsplyvayushchie-sobytiya-bubbling)

  
В этой модели сначала будет выполнен обработчик на элементе 3, затем на элементе 2, и последним будет выполнен обработчик на элементе 1.

Такой порядок называется "всплывающим", потому что событие поднимается с самых "глубоких" элементов в представлении DOM, к самым "верхним", как пузырек воздуха в воде.

Визуально это выглядит так (кликните на вложенном элементе, чтоб увидеть, какой будет порядок обработки события):

1

2

3

|  |  |
| --- | --- |
| 1 | <div class="d1" onclick="alert(1)"> |
| 2 | <div class="d2" onclick="alert(2)"> | |

|  |  |  |
| --- | --- | --- |
| 3 | <div class="d3" onclick="alert(3)"></div> | |
| 4 | </div> |

|  |  |
| --- | --- |
| 5 | </div> |

#### [**Остановка всплытия**](http://javascript.ru/tutorial/events/intro#ostanovka-vsplytiya)

Нужно понимать, что "всплытие" происходит всегда. При возникновении события на элементе, сигнал будет подниматься до самого высокого элемента, выполняя нужные обработчики.

Если какой-то обработчик хочет остановить всплытие и не выпускать событие дальше вверх - это делает следующий код:

|  |  |
| --- | --- |
| 01 | element.onclick = function(event) { |
| 02 | event = event || window.event // кросс-браузерно | |

|  |  |
| --- | --- |
| 03 |  |
| 04 | if (event.stopPropagation) { | |

|  |  |  |
| --- | --- | --- |
| 05 | // Вариант стандарта W3C: | |
| 06 | event.stopPropagation() |

|  |  |
| --- | --- |
| 07 | } else { |
| 08 | // Вариант Internet Explorer: | |

|  |  |  |
| --- | --- | --- |
| 09 | event.cancelBubble = true | |
| 10 | } |

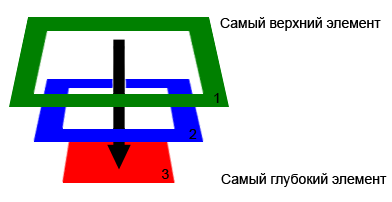
|  |  |
| --- | --- |
| 11 | } |

Можно уложить блок if/else в одну строчку:

|  |
| --- |
| event.stopPropagation ? event.stopPropagation() : (event.cancelBubble=true) |

### [Перехват событий (Capturing)](http://javascript.ru/tutorial/events/intro#perehvat-sobytiy-capturing)

Перехват - вторая, альтернативная всплытию модель порядка выполнения для события.

  
В этой модели сначала будет выполнен обработчик на элементе 1, затем - на элементе 2 и последним будет выполнен обработчик на элементе 3. Она называется "перехват", потому что родительские элементы могут обработать событие раньше, чем непосредственная цель события, как бы "перехватывая" обработку.

Визуально это выглядит так (кликните на вложенном элементе, чтоб увидеть, какой будет порядок обработки события, не поддерживается в IE):

1

2

3

|  |  |
| --- | --- |
| 01 | <div id="capt1" class="d1"> |
| 02 | <div id="capt2" class="d2"> | |

|  |  |  |
| --- | --- | --- |
| 03 | <div id="capt3" class="d3"></div> | |
| 04 | </div> |

|  |  |  |
| --- | --- | --- |
| 05 | </div> | |
| 06 |  |

|  |  |
| --- | --- |
| 07 | <script> |
| 08 | document.getElementById("capt1").addEventListener("click", function() { alert(1) }, true); | |

|  |  |
| --- | --- |
| 09 | document.getElementById("capt2").addEventListener("click", function() { alert(2) }, true); |
| 10 | document.getElementById("capt3").addEventListener("click", function() { alert(3) }, true); |

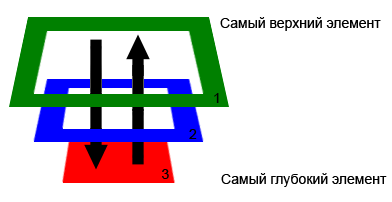
|  |  |
| --- | --- |
| 11 | </script> |

Такой порядок был предложен Netscape и никогда не поддерживался в Internet Explorer, поэтому в IE вы не сможете увидеть этот пример в действии.

Остальные браузеры поддерживают одновременно такой порядок и порядок всплытия, но из-за проблем с Internet Explorer де-факто его почти никто не использует.

### [Порядок обработки в стандарте W3C](http://javascript.ru/tutorial/events/intro#poryadok-obrabotki-v-standarte-w3c)

Решение от W3C объединяет обе модели: перехват и всплытие в одну универсальную.



При совершении действия, сначала событие перехватывается, пока не достигнет конечного элемента, затем всплывает.

Таким образом, разработчик сам решает, когда должен срабатывать обработчик события – при перехвате или при всплытии.

Визуально это выглядит так (кликните на вложенном элементе, чтоб увидеть, какой будет порядок обработки события, не для IE):

1

2

3

[Показать исходный код](http://javascript.ru/tutorial/events/intro#expandSource)

Если в качестве третьего параметра функции addEventListener передать значениеtrue, то событие будет срабатывать на фазе захвата, если false – то после окончания захвата, на фазе всплытия.

При установке обработчиков классическими методами (через свойство элемента или атрибут html тега) события всегда будут срабатывать на фазе всплытия.

Дальше мы вообще не будем рассматривать фазу захвата, так как в реальной жизни используется только всплытие.

## [Действие браузера по умолчанию](http://javascript.ru/tutorial/events/intro#deystvie-brauzera-po-umolchaniyu)

Браузер имеет своё собственное поведение по умолчанию для различных событий.

Например, клик по ссылке - сменить URL, клик правой кнопкой мыши - показать контекстное меню и т.п.

В ряде случаев реакцию браузера на событие можно убрать в обработчике.Для этого у IE и W3C есть, как и раньше, два по сути близких, но по виду разных способа:

|  |  |  |
| --- | --- | --- |
| 01 | element.onclick = function(event) { | |
| 02 | event = event || window.event |

|  |  |
| --- | --- |
| 03 |  |
| 04 | if (event.preventDefault) { | |

|  |  |  |
| --- | --- | --- |
| 05 | // Вариант стандарта W3C: | |
| 06 | event.preventDefault() |

|  |  |
| --- | --- |
| 07 | } else { |
| 08 | // Вариант Internet Explorer: | |

|  |  |  |
| --- | --- | --- |
| 09 | event.returnValue = false | |
| 10 | } |

|  |  |
| --- | --- |
| 11 | } |

Вместо if/else можно записать одну строчку:

|  |
| --- |
| .. |
| event.preventDefault ? event.preventDefault() : (event.returnValue=false) | |

|  |
| --- |
| ... |

Некоторые поведения по умолчанию происходят до вызова обработчика события. В этом случае их, конечно же, отменить нельзя.

Например, при фокусировке на ссылке - браузер выделяет ее пунктирной рамочкой.  
Это действие выполняется до события focus, поэтому отменить выделение в обработчике onfocus нельзя.

А переход по ссылке выполняется после события, поэтому его отменить можно.  
Обработчики onfocus и onclick на этой ссылке отменяют поведение по умолчанию:

[Кликни меня](http://javascript.ru/tutorial/events/intro)

При клике перехода не произойдет, а рамка вокруг ссылки появится.

Код примера:

|  |  |  |
| --- | --- | --- |
| 1 | var a = document.getElementById('my-focus-a') | |
| 2 | a.onfocus = a.onclick = function(e) { |

|  |  |
| --- | --- |
| 3 | e = e || window.event |
| 4 | // другая кроссбраузерная запись остановки события | |

|  |  |  |
| --- | --- | --- |
| 5 | e.preventDefault ? e.preventDefault() : (e.returnValue=false) | |
| 6 | } |

## [Смысл return false из обработчика](http://javascript.ru/tutorial/events/intro#smysl-return-false-iz-obrabotchika)

Возвращение return false из обработчика события предотвращает действие браузера по умолчанию, но не останавливает всплытие. В этом смысле следующие два кода эквивалентны:

|  |  |  |
| --- | --- | --- |
| 1 | function handler(event) { | |
| 2 | ... |

|  |  |  |
| --- | --- | --- |
| 3 | return false | |
| 4 | } |
| 1 | function handler(event) { | | |

|  |  |
| --- | --- |
| 2 | ... |
| 3 | if (event.preventDefault) { | |

|  |  |  |
| --- | --- | --- |
| 4 | event.preventDefault() | |
| 5 | } else { |

|  |  |  |
| --- | --- | --- |
| 6 | event.returnValue = false | |
| 7 | } |

|  |  |
| --- | --- |
| 8 | } |

Заметим, что хотя даже если всплытие и действия по умолчанию остановлены, но другие обработчики на текущем элементе все равно сработают.

|  |  |  |
| --- | --- | --- |
| 01 | elem = document.getElementById('TestStop') | |
| 02 |  |

|  |  |
| --- | --- |
| 03 | function handler(e) { |
| 04 | e.preventDefault() // браузер - стоять | |

|  |  |
| --- | --- |
| 05 | e.stopPropagation() // событие - не всплывать |
| 06 | return false // и вообще, мне больше ничего не надо | |

|  |  |  |
| --- | --- | --- |
| 07 | } | |
| 08 |  |

|  |  |  |
| --- | --- | --- |
| 09 | elem.addEventListener('click', handler, false) | |
| 10 |  |

|  |  |
| --- | --- |
| 11 | // следующий обработчик все равно сработает |
| 12 | elem.addEventListener('click', function() { alert('А я сработало..') }, false); | |

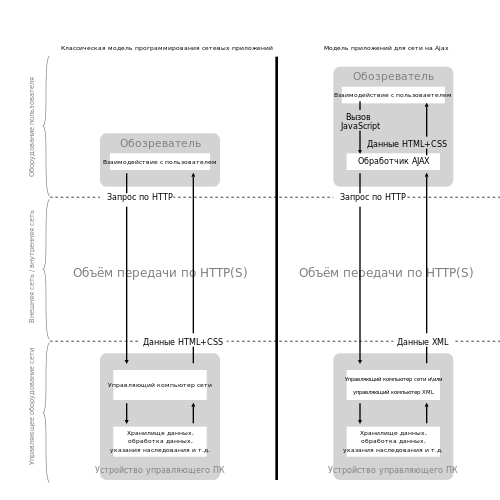
Проверить:

Действительно, браузер даже не гарантирует порядок, в котором сработают обработчики на одном элементе. Назначить можно в одном порядке, а сработают в другом.

Поэтому тем более один обработчик никак не может влиять на другие того же типа на том же элементе.

11. Ajax, Sockets

**AJAX**, Ajax ([ˈeɪdʒæks](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D0%B6%D0%B4%D1%83%D0%BD%D0%B0%D1%80%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D1%84%D0%BE%D0%BD%D0%B5%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B8%D0%B9_%D0%B0%D0%BB%D1%84%D0%B0%D0%B2%D0%B8%D1%82), от [англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Asynchronous Javascript and XML* — «асинхронный [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) и [XML](https://ru.wikipedia.org/wiki/XML)») — подход к построению интерактивных[пользовательских интерфейсов](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%BB%D1%8C%D0%B7%D0%BE%D0%B2%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D0%BA%D0%B8%D0%B9_%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81) [веб-приложений](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5), заключающийся в «фоновом» [обмене данными](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D0%BC%D0%B5%D0%BD_%D0%B4%D0%B0%D0%BD%D0%BD%D1%8B%D0%BC%D0%B8) [браузера](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D0%B1%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80) с [веб-сервером](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80). В результате, при обновлении данных [веб-страница](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D1%82%D1%80%D0%B0%D0%BD%D0%B8%D1%86%D0%B0) не перезагружается полностью, и веб-приложения становятся быстрее и удобнее.

[](https://commons.wikimedia.org/wiki/File:Ajax-model-ru.svg?uselang=ru)

Модель классических приложений для сети (слева) в прямом сравнении с применением AJAX (справа).

По-английски AJAX произносится как «э́йджэкс», по-русски довольно распространено «ая́кс».

## Содержание

  [[убрать](https://ru.wikipedia.org/wiki/AJAX)]

* [1Сравнение стандартного подхода и AJAX](https://ru.wikipedia.org/wiki/AJAX#.D0.A1.D1.80.D0.B0.D0.B2.D0.BD.D0.B5.D0.BD.D0.B8.D0.B5_.D1.81.D1.82.D0.B0.D0.BD.D0.B4.D0.B0.D1.80.D1.82.D0.BD.D0.BE.D0.B3.D0.BE_.D0.BF.D0.BE.D0.B4.D1.85.D0.BE.D0.B4.D0.B0_.D0.B8_AJAX)
* [2Технология](https://ru.wikipedia.org/wiki/AJAX#.D0.A2.D0.B5.D1.85.D0.BD.D0.BE.D0.BB.D0.BE.D0.B3.D0.B8.D1.8F)
* [3История](https://ru.wikipedia.org/wiki/AJAX#.D0.98.D1.81.D1.82.D0.BE.D1.80.D0.B8.D1.8F)
* [4Преимущества](https://ru.wikipedia.org/wiki/AJAX#.D0.9F.D1.80.D0.B5.D0.B8.D0.BC.D1.83.D1.89.D0.B5.D1.81.D1.82.D0.B2.D0.B0)
* [5Недостатки](https://ru.wikipedia.org/wiki/AJAX#.D0.9D.D0.B5.D0.B4.D0.BE.D1.81.D1.82.D0.B0.D1.82.D0.BA.D0.B8)
* [6Альтернативные технологии](https://ru.wikipedia.org/wiki/AJAX#.D0.90.D0.BB.D1.8C.D1.82.D0.B5.D1.80.D0.BD.D0.B0.D1.82.D0.B8.D0.B2.D0.BD.D1.8B.D0.B5_.D1.82.D0.B5.D1.85.D0.BD.D0.BE.D0.BB.D0.BE.D0.B3.D0.B8.D0.B8)
  + [6.1Adobe Flash](https://ru.wikipedia.org/wiki/AJAX#Adobe_Flash)
* [7См. также](https://ru.wikipedia.org/wiki/AJAX#.D0.A1.D0.BC._.D1.82.D0.B0.D0.BA.D0.B6.D0.B5)
* [8Примечания](https://ru.wikipedia.org/wiki/AJAX#.D0.9F.D1.80.D0.B8.D0.BC.D0.B5.D1.87.D0.B0.D0.BD.D0.B8.D1.8F)
* [9Литература](https://ru.wikipedia.org/wiki/AJAX#.D0.9B.D0.B8.D1.82.D0.B5.D1.80.D0.B0.D1.82.D1.83.D1.80.D0.B0)
* [10Ссылки](https://ru.wikipedia.org/wiki/AJAX#.D0.A1.D1.81.D1.8B.D0.BB.D0.BA.D0.B8)

## Сравнение стандартного подхода и AJAX[[править](https://ru.wikipedia.org/w/index.php?title=AJAX&veaction=edit&vesection=1) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=AJAX&action=edit&section=1)]

В классической модели веб-приложения:

* Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
* [Браузер](https://ru.wikipedia.org/wiki/%D0%91%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80) формирует и отправляет [запрос](https://ru.wikipedia.org/wiki/HTTP) [серверу](https://ru.wikipedia.org/wiki/%D0%92%D0%B5%D0%B1-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80).
* В ответ сервер генерирует совершенно новую веб-страницу и отправляет её браузеру и т. д. После чего браузер полностью перезагружает всю страницу.

При использовании AJAX:

* Пользователь заходит на веб-страницу и нажимает на какой-нибудь её элемент.
* [Скрипт](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D1%80%D0%B8%D0%BF%D1%82) (на языке [JavaScript](https://ru.wikipedia.org/wiki/JavaScript)) определяет, какая информация необходима для обновления страницы.
* [Браузер](https://ru.wikipedia.org/wiki/%D0%91%D1%80%D0%B0%D1%83%D0%B7%D0%B5%D1%80) отправляет соответствующий запрос на [сервер](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80).
* [Сервер](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80) возвращает только ту часть документа, на которую пришёл запрос.
* [Скрипт](https://ru.wikipedia.org/wiki/%D0%A1%D0%BA%D1%80%D0%B8%D0%BF%D1%82) вносит изменения с учётом полученной информации (без полной перезагрузки страницы).

## Технология[[править](https://ru.wikipedia.org/w/index.php?title=AJAX&veaction=edit&vesection=2) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=AJAX&action=edit&section=2)]

AJAX — не самостоятельная технология, а концепция использования нескольких смежных технологий. AJAX базируется на двух основных принципах:

* использование технологии динамического обращения к [серверу](https://ru.wikipedia.org/wiki/%D0%A1%D0%B5%D1%80%D0%B2%D0%B5%D1%80_(%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B5)) «на лету», без перезагрузки всей страницы полностью, например с использованием[XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest) (основной объект);
* через динамическое создание дочерних фреймов[[1]](https://ru.wikipedia.org/wiki/AJAX#cite_note-1);
* через динамическое создание тега <script>[[2]](https://ru.wikipedia.org/wiki/AJAX#cite_note-JsHttpRequest-2).
* через динамическое создание тега <img>, как это реализовано в Google Analytics.
* использование [DHTML](https://ru.wikipedia.org/wiki/DHTML) для динамического изменения содержания страницы;

Действия с интерфейсом преобразуются в операции с элементами [DOM](https://ru.wikipedia.org/wiki/Document_Object_Model) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Document Object Model*), с помощью которых обрабатываются данные, доступные пользователю, в результате чего представление их изменяется. Здесь же производится обработка перемещений и щелчков мышью, а также нажатий клавиш. Каскадные таблицы стилей, или CSS ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Cascading Style Sheets*), обеспечивают согласованный внешний вид элементов приложения и упрощают обращение к DOM-объектам. Объект XMLHttpRequest (или подобные механизмы) используется для асинхронного взаимодействия с сервером, обработки запросов пользователя и загрузки в процессе работы необходимых данных.

Три из этих четырёх технологий — [CSS](https://ru.wikipedia.org/wiki/CSS), DOM и [JavaScript](https://ru.wikipedia.org/wiki/JavaScript) — составляют [DHTML](https://ru.wikipedia.org/wiki/DHTML) ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *Dynamic HTML*). По мнению некоторых специалистов (книг),[[*каких?*](https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D0%BA%D0%B8%D0%BF%D0%B5%D0%B4%D0%B8%D1%8F:%D0%98%D0%B7%D0%B1%D0%B5%D0%B3%D0%B0%D0%B9%D1%82%D0%B5_%D0%BD%D0%B5%D0%BE%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D1%85_%D0%B2%D1%8B%D1%80%D0%B0%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9)]средства DHTML, появившиеся в [1997 году](https://ru.wikipedia.org/wiki/1997_%D0%B3%D0%BE%D0%B4), подавали большие надежды, но так и не оправдали их.

В качестве формата передачи данных могут использоваться фрагменты простого текста, [HTML](https://ru.wikipedia.org/wiki/HTML)-кода, [JSON](https://ru.wikipedia.org/wiki/JSON) или [XML](https://ru.wikipedia.org/wiki/XML).

Преимущества[[править](https://ru.wikipedia.org/w/index.php?title=AJAX&veaction=edit&vesection=4) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=AJAX&action=edit&section=4)]

**Экономия трафика**

Использование AJAX позволяет значительно [сократить трафик](https://ru.wikipedia.org/w/index.php?title=%D0%AD%D0%BA%D0%BE%D0%BD%D0%BE%D0%BC%D0%B8%D1%8F_%D1%82%D1%80%D0%B0%D1%84%D0%B8%D0%BA%D0%B0&action=edit&redlink=1) при работе с веб-приложением благодаря тому, что вместо загрузки всей страницы достаточно загрузить только изменившуюся часть, или вообще только получить/передать набор данных в формате [JSON](https://ru.wikipedia.org/wiki/JSON) или [XML](https://ru.wikipedia.org/wiki/XML), а затем изменить содержимое страницы с помощью JavaScript.

**Уменьшение нагрузки на сервер**

При правильной реализации, AJAX позволяет снизить нагрузку на сервер в несколько раз.

В частности, все страницы сайта чаще всего генерируются по одному шаблону, включая неизменные элементы («шапка», «навигационная панель», «подвал» и т. д.) для генерации которых требуются обращения к разным файлам, время на обработку скриптов (а иногда и запросы к БД) — всё это можно опустить, если заменить полную загрузку страницы генерацией и передачей лишь содержательной части. Дизайн страницы также обычно содержит множество файлов, связанных с оформлением (картинки, стили), на повторную обработку которых не надо тратить время, используя AJAX (экономия на количестве HTTP-соединений значительно выгоднее, чем на сокращении трафика каждого из них).

**Ускорение реакции интерфейса**

Поскольку загрузка изменившейся части значительно быстрее, то пользователь видит результат своих действий быстрее и без мерцания страницы (возникающего при полной перезагрузке).

**Почти безграничные возможности для интерактивной обработки**

Например, при вводе поискового запроса в [Google](https://ru.wikipedia.org/wiki/Google) выводится подсказка с возможными вариантами запроса. На многих сайтах при регистрации пользователь вводит имя, и сразу же видит, доступно это имя или нет. AJAX удобен для программирования [чатов](https://ru.wikipedia.org/wiki/%D0%A7%D0%B0%D1%82), [административных панелей](https://ru.wikipedia.org/w/index.php?title=%D0%90%D0%B4%D0%BC%D0%B8%D0%BD%D0%B8%D1%81%D1%82%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BD%D0%B5%D0%BB%D1%8C&action=edit&redlink=1) и других инструментов, которые выводят меняющиеся со временем данные.

Недостатки[[править](https://ru.wikipedia.org/w/index.php?title=AJAX&veaction=edit&vesection=5) | [править вики-текст](https://ru.wikipedia.org/w/index.php?title=AJAX&action=edit&section=5)]

**Отсутствие интеграции со стандартными инструментами браузера**

Динамически создаваемые страницы не регистрируются браузером в истории посещения страниц, поэтому не работает кнопка «Назад», предоставляющая пользователям возможность вернуться к просмотренным ранее страницам, но существуют скрипты, которые могут решить эту проблему.

Другой недостаток изменения содержимого страницы при постоянном [URL](https://ru.wikipedia.org/wiki/URL) заключается в невозможности сохранения закладки на желаемый материал. Проблему можно успешно решить с помощью History.pushState[[4]](https://ru.wikipedia.org/wiki/AJAX#cite_note-4).

**Динамически загружаемое содержимое не доступно поисковикам (если не проверять запрос, обычный он или**[**XMLHttpRequest**](https://ru.wikipedia.org/wiki/XMLHttpRequest)**)**

Поисковые машины не могут выполнять [JavaScript](https://ru.wikipedia.org/wiki/JavaScript), поэтому разработчики должны позаботиться об альтернативных способах доступа к содержимому сайта.

**Старые методы учёта статистики сайтов становятся неактуальными**

Многие сервисы статистики ведут учёт просмотров новых страниц сайта. Для сайтов, страницы которых широко используют AJAX, такая статистика теряет актуальность.

**Усложнение проекта**

Перераспределяется логика обработки данных — происходит выделение и частичный перенос на сторону клиента процессов первичного форматирования данных. Это усложняет контроль целостности форматов и типов. Конечный эффект технологии может быть нивелирован необоснованным ростом затрат на кодирование и управление проектом, а также риском снижения доступности сервиса для конечных пользователей.

**Требуется включенный JavaScript в браузере**

JavaScript может быть выключен из соображений безопасности. И, конечно же, AJAX-страницы труднодоступны неполнофункциональным браузерам, роботам и веб-архивам.

**Низкая скорость при грубом программировании**

Казалось бы, AJAX предназначен именно для повышения скорости. Но, когда AJAX-запросов на одной странице много и, например, по каждому щелчку подгружается список, AJAX-страница становится даже медленнее традиционной.

**Риск**[**фабрикации запросов другими сайтами**](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%B4%D0%B5%D0%BB%D0%BA%D0%B0_%D0%BC%D0%B5%D0%B6%D1%81%D0%B0%D0%B9%D1%82%D0%BE%D0%B2%D1%8B%D1%85_%D0%B7%D0%B0%D0%BF%D1%80%D0%BE%D1%81%D0%BE%D0%B2)

Результат работы AJAX-запроса может являться [JavaScript](https://ru.wikipedia.org/wiki/JavaScript)-кодом (в частности, [JSON](https://ru.wikipedia.org/wiki/JSON)). [XMLHttpRequest](https://ru.wikipedia.org/wiki/XMLHttpRequest) действует только [в пределах одного домена](https://ru.wikipedia.org/wiki/Same_Domain_Policy), а вот тег <script> — нет. Если написать

**WebSocket** — протокол [полнодуплексной связи](https://ru.wikipedia.org/wiki/%D0%94%D1%83%D0%BF%D0%BB%D0%B5%D0%BA%D1%81_(%D1%82%D0%B5%D0%BB%D0%B5%D0%BA%D0%BE%D0%BC%D0%BC%D1%83%D0%BD%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D0%B8)) поверх [TCP](https://ru.wikipedia.org/wiki/TCP)-соединения, предназначенный для обмена сообщениями между браузером и веб-сервером в режиме реального времени.

В настоящее время в [W3C](https://ru.wikipedia.org/wiki/W3C) осуществляется стандартизация API Web Sockets. Черновой вариант стандарта этого протокола утверждён [IETF](https://ru.wikipedia.org/wiki/IETF).

WebSocket разработан для имплементирования в web-браузерах и web-серверах, но он может быть использован для любого клиентского или серверного приложения. Протокол WebSocket независимый протокол основанный на протоколе TCP. Этот протокол делает возможным более тесное взаимодействие между браузером и web сайтом для управления, способствуя распространению live content и созданию real-time игр.