

# Современный учебник JavaScript

© Илья Кантор

Сборка от 23 июля 2013 для чтения с устройств

Внимание, эта сборка может быть устаревшей и не соответствовать текущему тексту.

Актуальный онлайн-учебник, с интерактивными примерами, доступен по адресу <http://learn.javascript.ru>.

Вопросы по JavaScript можно задавать в комментариях на сайте или на форуме [javascript.ru/forum](http://javascript.ru/forum).

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу [iliakan@javascript.ru](mailto:iliakan@javascript.ru).

## Глава: Оптимизация

В файле находится только одна глава учебника. Это сделано в целях уменьшения размера файла, для удобного чтения с устройств.

## Содержание

### Управление памятью в JS и DOM

- Управление памятью в JavaScript

  - Достижимость и наличие ссылок

- Иллюстрация: объект

  - Иллюстрация: удаление ссылки

- Иллюстрация: круговые ссылки

  - Иллюстрация: недостижимый остров

- Замыкания

- Очистка памяти при работе с DOM

  - Удаление `removeChild` без ссылок

Удаление, если есть ссылка

Удаление через innerHTML

setInterval/setTimeout, XMLHttpRequest

Влияние управления памятью на скорость

## Утечки памяти

Коллекция утечек в IE

Утечка DOM ↔ JS в IE<8

Утечка DOM ↔ JS, вариант для IE<9

Утечка IE8 при обращении к коллекциям таблицы

Утечка через XMLHttpRequest в IE<9

Объемы утечек памяти

jQuery: утечки и борьба с утечками

Примеры утечек в jQuery

Используем jQuery без утечек

Улучшение производительности jQuery

Поиск и устранение утечек памяти

Проверка на утечки

Настройка браузера

Инструменты

---

# Управление памятью в JS и DOM

Управление памятью обычно незаметно. Мы создаём объекты, элементы, обработчики.. Всё это занимает память.

Что происходит с объектом, когда он становится «не нужен»? Как освобождается память из-под удалённых DOM-элементов? Возможно ли «переполнение» памяти? Для ответа на эти вопросы — залезем «под капот».

## Управление памятью в JavaScript

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

1. Определённое множество значений считается достижимым изначально, в частности:
  - Значения, ссылки на которые содержатся в стеке вызова, то есть — все локальные переменные и параметры функций, которые в настоящий момент вызываются.
  - Все глобальные переменные.Эти значения гарантированно хранятся в памяти. Мы будем называть их «корнями».
2. Любое другое значение сохраняется в памяти лишь до тех пор, пока доступно из корня по ссылке или цепочке ссылок.

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

Далее мы посмотрим ряд примеров, которые помогут в этом разобраться.

## Достижимость и наличие ссылок

Можно сказать проще: «значение остаётся в памяти, пока на него есть ссылка». Это частично верно, но не совсем.

- **Верно** — в том плане, что если на значение не остаётся ссылок, то память из-под него очищается.

Например, было значение в переменной, и эту переменную перезаписали:

```
var str = "Моя Строка";  
str = null;
```

Теперь значение "Моя Строка" больше недоступно. Память будет освобождена.

В простейших случаях, вроде такого, освобождение памяти может происходить тут же.

- **Неверно** — так как ссылки могут быть, но при этом значение

недостижимо, и его нужно удалять из памяти.

Такая ситуация возникает с объектами, при наличии ссылок друг на друга:

```
1 var obj1 = {};  
2 var obj2 = {};  
3 obj1.a = obj2;  
4 obj2.b = obj1;  
5  
6 obj1 = obj2 = null;
```

Несмотря на то, что на каждый из объектов `obj1`, `obj2` в примере выше есть ссылка от «собрата», последняя строка делает эти объекты в совокупности недостижимыми.

Поэтому они будут удалены.

Чтобы отследить такие сложные случаи, и придуман [сборщик мусора](#), который время от времени перебирает объекты и ищет недоступные, с использованием хитрых алгоритмов и оптимизаций.

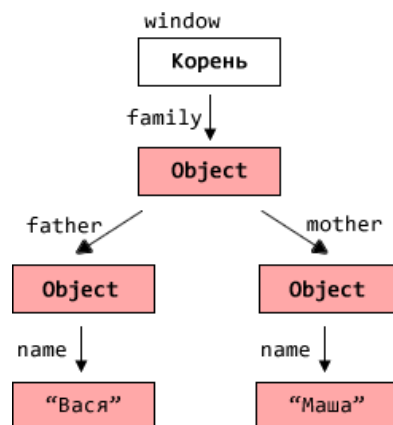
## Иллюстрация: объект

Рассмотрим пример объекта:

Код

```
1 var family = { };  
2  
3 family.father = {  
4   name: "Вася"  
5 };  
6  
7 family.mother = {  
8   name: "Маша"  
9 };
```

Структура в памяти



Этот код создаёт объект `family` и два дополнительных объекта, доступных по ссылкам `family.father` и `family.mother`.

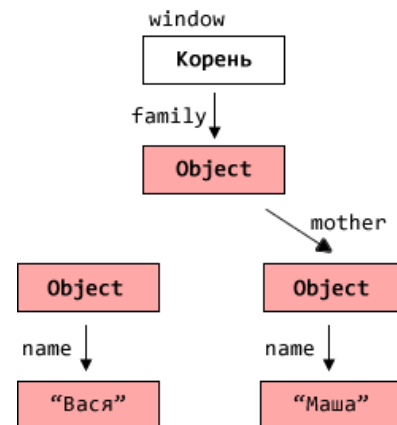
Иллюстрация: удаление ссылки

Теперь посмотрим, что будет, если удалить ссылку `family.father` при помощи `delete`:

## Код

```
01 var family = { };
02
03 family.father = {
04   name: "Вася"
05 };
06
07 family.mother = {
08   name: "Маша"
09 };
10
11 delete family.father;
```

## Структура в памяти



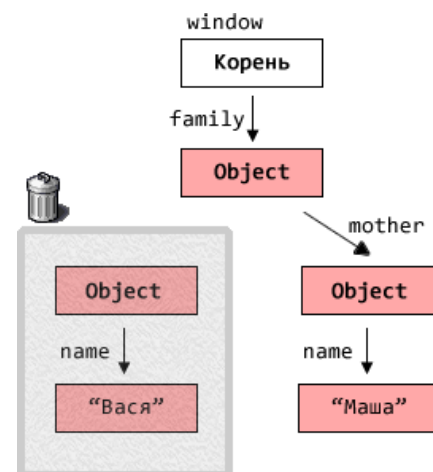
На объект `family.father` больше нет ссылок. Он стал недостижимым и будет удалён.

Данные «к удалению» на картинке ниже окрашены серым:

## Код

```
01 var family = {
02   father: {
03     name: "Вася"
04   },
05
06   mother: {
07     name: "Маша"
08   }
09 };
10
11 delete family.father;
```

## Структура в памяти

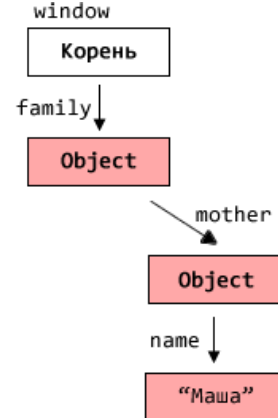


После того, как сработает сборщик мусора, картина в памяти будет такой:

## Код

```
01 var family = {
02   father: {
03     name: "Вася"
04   },
05
```

## Структура в памяти



```

06   mother: {
07       name: "Маша"
08   }
09 };
10
11 delete family.father;
  
```

## Иллюстрация: круговые ссылки

Рассмотрим такой же код, но с дополнительными ссылками.

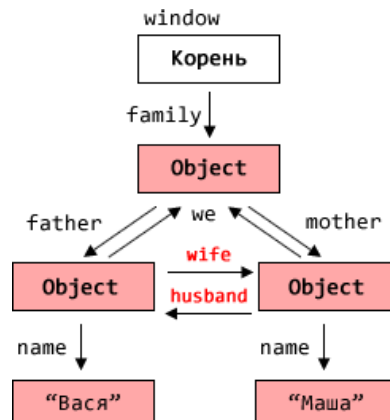
Пусть внутренние объекты ссылаются друг на друга:

Код

Структура в памяти

```

01 var family = {
02     father: {
03         name: "Вася"
04     },
05
06     mother: {
07         name: "Маша"
08     }
09 };
10
11 // добавим перекрёстных ссылок
12 family.father.wife = family.mother;
13 family.mother.husband = family.father;
14 family.father.we = family;
15 family.mother.we = family;
  
```



Теперь при удалении той же ссылки `family.father` все объекты по-прежнему остаются **достижимыми**!

Чтобы его удалить, нужно дополнительно «обрезать» ссылку `family.mother.husband`.

Иллюстрация этого показана на рисунке ниже.

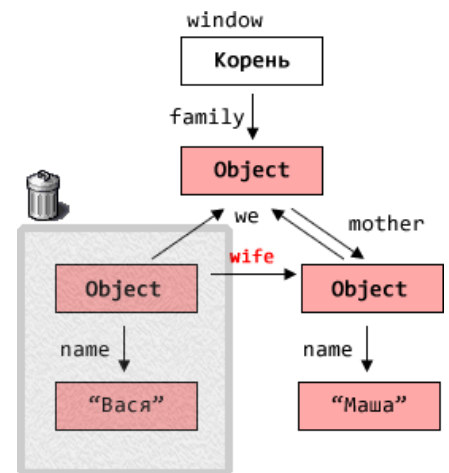
Код

Структура в памяти

```

01 var family = {
02   father: {
03     name: "Вася"
04   },
05
06   mother: {
07     name: "Маша"
08   }
09 };
10
11 family.father.wife = family.mother;
12 family.mother.husband = family.father;
13 family.father.we = family;
14 family.mother.we = family;
15
16 delete family.father;
17 delete family.mother.husband;

```



Обратим внимание — из удаляемого объекта выходят ссылки. Но это ничего не значит. Важны лишь те, которые входят, то есть по которым объект может быть достижим.

## Иллюстрация: недостижимый остров

Всё «семейство» объектов, которое мы рассматривали выше, достижимы исключительно через `window.family`.

Если записать в `window.family` что-то ещё, то все они, вместе со своими внутренними ссылками станут «недостижимым островом» и будут удалены:

### Код

```

01 var family = {
02   father: {
03     name: "Вася"
04   },
05
06   mother: {
07     name: "Маша"
08   }
09 };
10

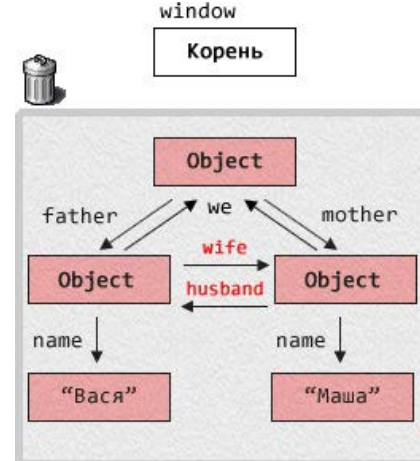
```

### Структура в памяти

```

11 family.father.wife = family.mother;
12 family.mother.husband = family.father;
13 family.father.we = family;
14 family.mother.we = family;
15
16 family = null;

```



## Замыкания

Замыкания следуют тем же правилам, что и обычные объекты.

Если внутренняя функция остаётся доступной, после того как отработала внешняя, то она сохраняет ссылку на её объект переменных.

Соответственно, сохраняются и все объекты, достижимые из них.

Например:

```

01 function makeUser(userName) {
02
03     var user = { };
04
05     user.created = new Date();
06
07     user.sayHi = function() {
08         alert(userName);
09     };
10
11     return user;
12 }
13
14 var vasya = makeUser("Вася");
15 vasya.sayHi();

```

Функция sayHi достижима из объекта vasya, а поскольку она ссылается на объект с переменными makeUser, то и userName останется в памяти.

А если запустить delete vasya.sayHi, то функция перестанет быть



достижимой и на объект с переменными больше не останется ссылок, так что он будет удалён.

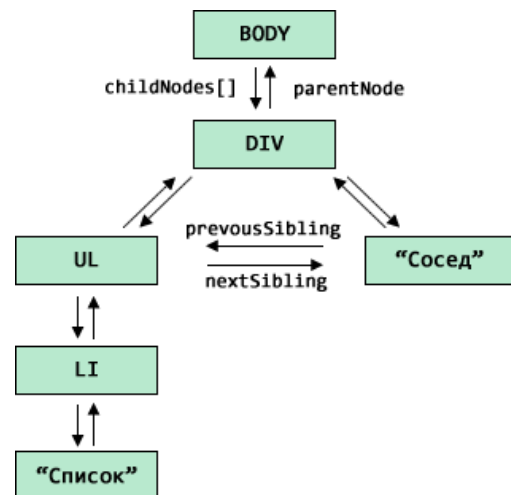
## Очистка памяти при работе с DOM

Для примера рассмотрим следующий HTML:

Код

Структура в памяти

```
01 <html>
02 <body>
03   <div>
04     <ul>
05       <li>Список</li>
06     </ul>
07     Сосед
08   </div>
09 </body>
10 </html>
```



### Удаление removeChild без ссылок

Что произойдёт с памятью при удалении элемента?

Например, удалим DIV из BODY:

```
document.body.removeChild(document.body.children[0]);
```

Операция removeChild разрывает связи между DIV и его родителем BODY.

В результате вся нижележащая структура DOM оказывается недостижимой. Память будет очищена.

### Удаление, если есть ссылка

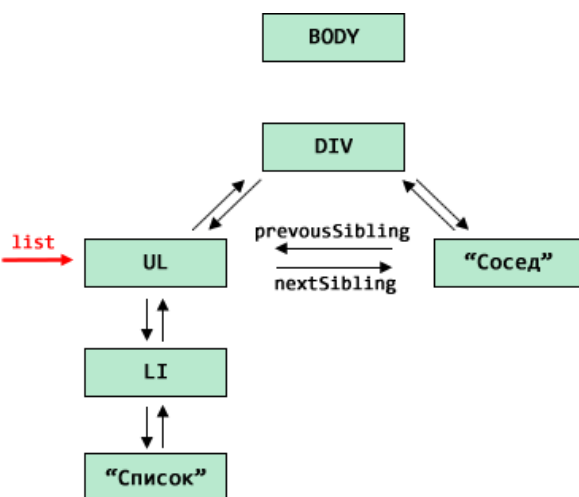
Например, у нас есть компонент на JavaScript, из которого есть ссылка.

Пусть это будет переменная, которая ссылается на UL:

```
var list = document.getElementsByTagName('UL')[0];
document.body.removeChild(document.body.children[0]);
```

**В этом случае, так как DOM взаимосвязан, то он полностью остаётся**

**в памяти!** Включая детей, родителей, соседей и т.п... Всё из-за внешней ссылки `list`, которая делает их достижимыми.



То есть, в этом случае DOM работает по той же логике, что и обычные объекты.

## Удаление через `innerHTML`

..А вот здесь нас встретит небольшой браузерный «зоопарк». Дело в том, что удаление через очистку `elem.innerHTML=""` браузеры интерпретируют по-разному.

По идее, при присвоении `innerHTML` из DOM должны удаляться текущие узлы и добавляться новые (из html). Но стандарт ничего не говорит о том, что делать с узлами после удаления. И тут браузеры чудят каждый по-своему.

Посмотрим, что произойдёт с DOM-структурой при очистке `BODY`, если на какой-либо элемент есть ссылка.

```
var list = document.getElementsByTagName('UL')[0];
document.body.innerHTML = "";
```

Обращаю внимание — связь разрывается только между `DIV` и `BODY`, т.е. на верхнем уровне, а `list` — это произвольный элемент.

Чтобы увидеть, что останется в памяти, а что нет — запустим код:

```
01 <div>
02   <ul>
03     <li>Список</li>
04   </ul>
```

```

05     Сосед
06 </div>
07
08 <script>
09     var list = document.getElementsByTagName('ul')[0];
10     document.body.innerHTML = ''; // удалили DIV
11
12     alert(list.parentNode); // целая ли ссылка UL -> DIV ?
13     alert(list.nextSibling); // живы ли соседи UL ?
14     alert(list.children.length); // живы ли потомки UL ?
15 </script>

```

Как ни странно, браузеры ведут себя по-разному:

	parentNode	nextSibling	children.length
Chrome/Safari	null	null	1
Firefox	Элемент	Элемент	1
Opera	Элемент	Элемент	1
IE 6-9	null	null	0

Иными словами, браузеры ведут себя с различной степенью агрессивности по отношению к элементам.

## Firefox, Opera

Главные пацифисты. Оставляют всё, на что есть ссылки, т.е. элемент, его родителя, соседей и детей.

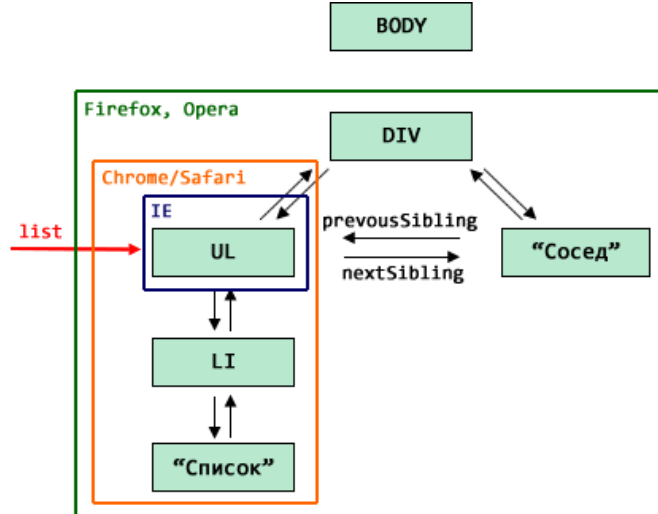
## Chrome/Safari

Считают, что раз мы задали ссылку на UL, то нам нужно только это поддерево, а остальные (соседи, родитель) можно удалить.

## Internet Explorer

Как ни странно, самый агрессивный. Удаляет вообще всё, кроме узла, на который есть ссылка. Это поведение одинаково для всех версий IE.

На иллюстрации ниже показано, какую часть DOM оставит каждый из



браузеров:

Таким образом, кросс-браузерно, при поддержке IE, гарантировано одно: **сам узел, на который есть ссылка, останется в памяти.**

Это поведение специфично для `innerHTML`. При обычном удалении узла из DOM все браузеры ведут себя как Firefox/Opera, т.е. остаётся всё.

## setInterval/setTimeout, XMLHttpRequest

Кроме *достижимости* через глобальные переменные, есть ещё и *достижимость* через неявные ссылки.

Например, при передаче функции в `setInterval/setTimeout` создаётся внутренняя ссылка на неё, через которую браузер её будет запускать. Поэтому функция остаётся жить, даже если других ссылок на неё нет.

```
// Функция будет жить в памяти, пока не сработал (или не
очищен) таймер
setTimeout(function() {}, 100);
```

→ Для `setTimeout` — внутренняя ссылка исчезнет после исполнения функции.

→ Для `setInterval` — ссылка исчезнет при очистке таймера.

Аналогично работает `XMLHttpRequest` — при отправке запроса методом `send` (не раньше!) браузер создаёт внутреннюю ссылку на него. При получении ответа (`readyState == 4`) — удаляет ссылку, и если объект не достижим из других мест — память под ним будет освобождена.

## Влияние управления памятью на скорость

На создание новых объектов и их удаление тратится время. Это всегда стоит иметь в виду.

В качестве примера рассмотрим рекурсию.

При вызове каждой функции создаются вспомогательные объекты (arguments, LexicalEnvironment). Поэтому рекурсивный код будет всегда медленнее использующего цикл.

Пример ниже тестирует сложение чисел до данного через рекурсию по сравнению с обычным циклом:

```
01 function sumTo(n) {    // обычный цикл 1+2+...+n
02     var result = 0;
03     for (var i=1; i<=n; i++) {
04         result += i;
05     }
06     return result;
07 }
08
09 function sumToRec(n) { // рекурсия sumToRec(n) =
    n+sumToRec(n-1)
10     return n == 1 ? 1 : n + sumToRec(n-1);
11 }
12
13 var d = new Date;
14 for (var i=1;i<1000;i++) sumTo(1000); // цикл
15 var timeLoop = new Date - d;
16
17 var d = new Date;
18 for (var i=1;i<1000;i++) sumToRec(1000); // рекурсия
19 var timeRecursion = new Date - d;
20
21 alert("Разница в "+(timeRecursion/timeLoop)+" раз");
```

Различие в скорости на таком примере, когда сама функция почти ничего не делает, может составлять 4-10 раз.

## Утечки памяти

*Утечки памяти* происходят, когда браузер по какой-то причине не может освободить память от недостижимых объектов.

Обычно это происходит автоматически ([Управление памятью в JS и DOM](#)). Кроме того, браузер освобождает память при переходе на другую страницу. Поэтому утечки в реальной жизни проявляют себя в двух ситуациях:

1. Приложение, в котором посетитель все время на одной странице и работает со сложным JavaScript-интерфейсом. В этом случае утечки могут постепенно съедать доступную память.
2. Страница регулярно делает что-то, вызывающее утечку памяти. Посетитель (например, менеджер) оставляет компьютер на ночь включенным, чтобы не закрывать браузер с кучей вкладок. Приходит утром — а браузер съел всю память и рухнул и сильно тормозит.

Утечки бывают из-за ошибок браузера, ошибок в расширениях браузера и, гораздо реже, по причине ошибок в архитектуре JavaScript-кода. Мы разберём несколько наиболее частых и важных примеров.

## Коллекция утечек в IE

### Утечка DOM JS в IE<8

**IE до версии 8 не умел очищать циклические ссылки, появляющиеся между DOM-объектами и объектами JavaScript. В результате и DOM и JS оставались в памяти навсегда.**

 Пропустите эту секцию, если IE<9 не нужен

Проблема была особенно серьезна в IE6 до SP3 (или до обновления июня 2007 года), там память не освобождалась даже при переходе на другую страницу.

Сейчас она существует в IE6, 7, а также, в облегчённом варианте, в IE8 (см. далее). Если вы не поддерживаете IE<9, то можете пропустить эту секцию.

Функция `setHandler` в примере ниже ведёт к утечке памяти в IE6,7:

```

1 function setHandler() {
2   var elem = document.getElementById('id');
3   elem.onclick = function() {
4     /* может быть пустая функция, не важно */
5   };
6 }

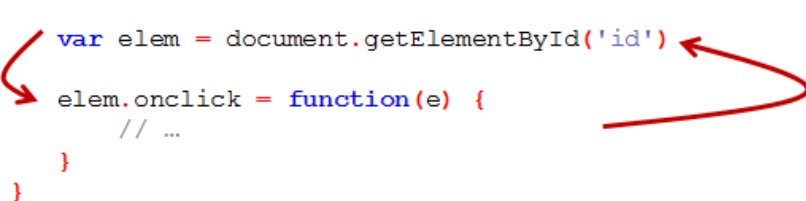
```

Элемент `elem` здесь ссылается на JavaScript-функцию через ссылку `onclick` напрямую, через свойство, а функция ссылается на `elem` через замыкание.

```

function setHandler() {
  var elem = document.getElementById('id')
  elem.onclick = function(e) {
    // ...
  }
}

```



A red arrow points from the `elem` variable to the `function` object, and another red arrow points from the `function` object back to the `elem` variable, forming a cycle.

Здесь вместо DOM-элемента в IE может быть XMLHttpRequest, ActiveX, любой другой COM-объект. Круговая ссылка гарантирует утечку.

**Обойти утечки памяти в IE можно, разорвав циклические ссылки.**

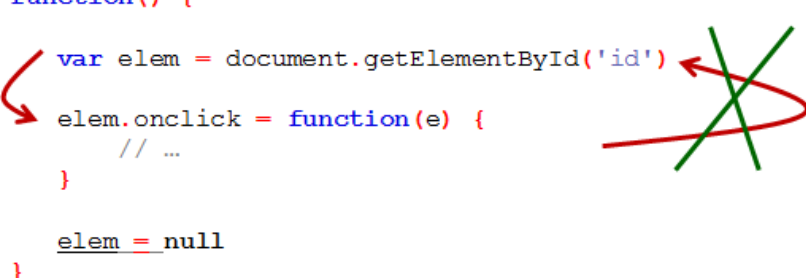
Например, можно удалить ссылку на `elem` из замыкания, присвоив `elem = null`. Таким образом обработчик больше не ссылается на DOM-элемент. Циклическая ссылка разорвана:

```

function() {
  var elem = document.getElementById('id')
  elem.onclick = function(e) {
    // ...
  }

  elem = null
}

```



The same code as before, but with `elem = null` added at the end of the function. A red arrow still points from `elem` to the `function`, but a green 'X' is drawn over the return arrow, indicating the cycle is broken.



Больше информации об этой утечке вы можете почерпнуть из статей: [Understanding and Solving Internet Explorer Leak Patterns](#) и [Circular Memory Leak Mitigation](#).

## Утечка DOM JS, вариант для IE<9

В браузере IE8 была проведена серьёзная работа над ошибками, и пример, описанный выше, больше не приводит к утечке.

Но ситуация исправлена не до конца. Утечка в IE<9 появляется, если круговая ссылка возникает «через объект».

Чтобы было понятнее, о чём речь, посмотрите на следующий код. Он вызывает утечку памяти:

```
01 function leak() {
02     // Создаём новый DIV, добавляем к BODY
03     var elem = document.createElement('div');
04     document.body.appendChild(elem);
05
06     // Записываем в свойство жирный объект
07     elem.__expando = {
08         bigAss: new Array(1000000).join('lalala')
09     };
10
11     // Создаём круговую ссылку. Без этой строки утечки не
12     // будет.
13     elem.__expando.__elem = elem;
14
15     // Удалить элемент из DOM. Браузер должен очистить
16     // память.
17     elem.parentElement.removeChild(elem);
18 }
```

[Открыть в новом окне \(в IE\)](#)

Этот пример, течёт в IE6,7,8, а также в IE9 в режиме совместимости с IE8. Проблема — в круговой ссылке `elem.__expando__elem = elem`.

Утечка может возникать и неявным образом, через замыкание:

```
01 function leak() {
02     var elem = document.createElement('div');
03     document.body.appendChild(elem);
04
05     elem.__expando = {
06         bigAss: new Array(1000000).join('lalala'),
07         method: function() {} // создаётся круговая ссылка
08         // через замыкание
09     };
10
11     // Удалить элемент из DOM. Браузер должен очистить
12     // память.
13     elem.parentElement.removeChild(elem);
14 }
```



[Открыть в новом окне \(в IE\)](#)

**Без метода `method` здесь утечки не возникнет.**

Бывает ли такая ситуация в реальной жизни? Или это — целиком синтетический пример, для заумных программистов?

Да, конечно бывает. Например, при разработке компоненты.

Во-первых, сам объект компоненты хранит ссылку на её DOM-элемент.

Например:

```
1 function Menu(elem) {  
2   elem.onclick = function() {};  
3 }  
4  
5 var menu = new Menu(elem); // menu содержит ссылку на elem
```

То есть, компонент всегда знает свой элемент.

Но бывают ситуации, когда нужно пойти в обратном направлении, а именно — по элементу определить, какой на нём компонент. Например, чтобы при делегировании, чтобы передать обработку события методу виджета. При Drag'n'Drop, чтобы получить компонент, над которым проносится элемент.

Сама задача не является чем-то из ряда вон выходящим. Вполне естественно, что JS-компонент привязан к элементу, а элемент знает о компоненте на нём. Но в IE<9 прямая привязка ведёт к утечке памяти!

```
1 function Menu(elem) {  
2   elem.onclick = function() {};  
3 }  
4  
5 var menu = new Menu(elem); // Menu содержит ссылку на elem  
6 elem.menu = menu; // вот такая привязка или что-то  
  подобное ведёт к утечке
```

[Открыть в новом окне \(в IE\)](#)

Такая привязка удобна, т.к. мы по DOM-элементу можем получить JS-компонент, который к нему привязан. Но, как видим, ведёт к утечке в IE<9.

## Утечка IE8 при обращении к коллекциям таблицы

Эта утечка происходит только в IE8 в стандартном режиме. В нём при обращении к табличным псевдо-массивам (напр. `rows`) создаются и не очищаются внутренние ссылки, что приводит к утечкам.

Также воспроизводится в новых IE в режиме совместимости с IE8.

Код:

```
01 var elem = document.createElement('div'); // любой элемент
02
03 function leak() {
04
05     elem.innerHTML = '<table><tr><td>1</td></tr></table>';
06
07     elem.firstChild.rows[0]; // просто доступ через rows[]
    приводит к утечке
08     // при том, что мы даже не сохраняем значение в
    переменную
09
10     elem.removeChild(elem.firstChild); // удалить таблицу
    (*)
11     // alert(elem.childNodes.length) // выдал бы 0, elem
    очищен, всё честно
12 }
```

[Открыть в новом окне \(в IE\)](#)

Особенности:

- Если убрать отмеченную строку, то утечки не будет.
- Если заменить строку (\*) на `elem.innerHTML = ''`, то память будет очищена, т.к. этот способ работает по-другому, нежели просто `removeChild` (см. главу [Управление памятью в JS и DOM](#)).
- Утечка произойдёт не только при доступе к `rows`, но и к другим свойствам, например `elem.firstChild.tBodies[0]`.

Эта утечка проявляется, в частности, при удалении детей элемента следующей функцией:

```
function empty(elem) {  
    while(elem.firstChild) elem.removeChild(elem.firstChild);  
}
```

Если идёт доступ к табличным коллекциям и регулярное обновление таблиц при помощи DOM-методов — утечка в IE8 будет расти.

Более подробно вы можете почитать об этой утечке в статье [Утечки памяти в IE8](#), или [страшная сказка со счастливым концом](#).

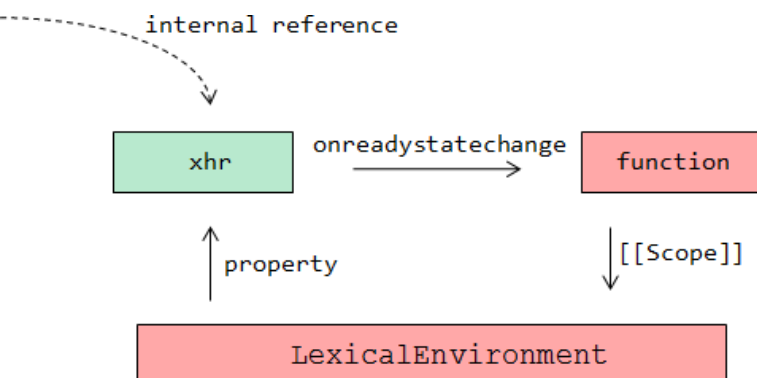
## Утечка через XMLHttpRequest в IE<9

Следующий код вызывает утечки памяти в IE<9:

```
01 function leak() {  
02     var xhr = new XMLHttpRequest(); // в IE6 создать через  
    ActiveX  
03  
04     xhr.open('GET', '/server.do', true);  
05  
06     xhr.onreadystatechange = function() {  
07         if(xhr.readyState == 4 && xhr.status == 200)  
08             // ...  
09         }  
10     }  
11  
12     xhr.send(null);  
13 }
```

Как вы думаете, почему? Если вы внимательно читали то, что написано выше, то имеете информацию для ответа на этот вопрос..

Посмотрим, какая структура памяти создается при каждом запуске:



Когда запускается асинхронный запрос `xhr`, браузер создаёт специальную

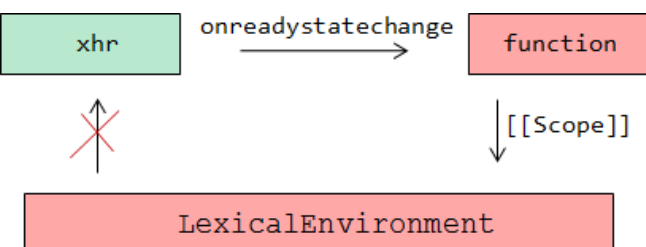
внутреннюю ссылку (internal reference) на этот объект. находится в процессе коммуникации. Именно поэтому объект xhr будет жив после окончания работы функции.

Когда запрос завершен, браузер удаляет внутреннюю ссылку, xhr становится недостижимым и память очищается... Везде, кроме IE<9.

[Открыть в новом окне \(в IE<9\)](#) (откройте страницу и пусть поработает минут 20 - съест всю память, включая виртуальную).

Чтобы это исправить, нам нужно разорвать круговую ссылку XMLHttpRequest ↔ JS. Например, можно удалить xhr из замыкания:

```
01 function leak() {
02   var xhr = new XMLHttpRequest();
03
04   xhr.open('GET', 'something.js?' + Math.random(), true);
05
06   xhr.onreadystatechange = function() {
07     if (xhr.readyState != 4) return;
08
09     if (xhr.status == 200) {
10       document.getElementById('test').innerHTML++;
11     }
12
13     xhr = null; // по завершении запроса удаляем ссылку из
14     // замыкания
15   }
16   xhr.send(null);
17 }
```



Теперь циклической ссылки нет — мы устранили утечку.

[Посмотреть исправленный пример для IE в отдельном окне.](#)

## Объемы утечек памяти

Объем «утекающей» памяти может быть небольшим. Тогда это почти

не ощущается. Но так как замыкания ведут к сохранению переменных внешних функций, то одна функция может тянуть за собой много чего ещё.

Представьте, вы создали функцию, и одна из ее переменных содержит очень большую по объему строку (например, получает с сервера).

```
01 function f() {  
02     var data = "Большой объем данных, например, переданных сервером"  
03  
04     /* делаем что-то хорошее (ну или плохое) с полученными данными */  
05  
06     function inner() {  
07         // ...  
08     }  
09  
10     return inner;  
11 }
```

Пока функция `inner` остается в памяти, `LexicalEnvironment` с переменной большого объема внутри висит в памяти.

Висит до тех пор, пока функция `inner` жива.

Интерпретатор JavaScript не знает, какие из переменных функции `inner` будут использованы, поэтому оставляет их все.

То есть, код может быть спроектирован так, что никакой утечки нет. По вполне разумной причине может создаваться множество функций, а память будет расти потому, что функция тянет за собой своё замыкание.

Сэкономить память здесь вполне можно. Мы же знаем, что переменная `data` не используется в `inner`. Поэтому просто обнулим её:

```
01 function f() {  
02     var data = "Большое количество данных, например, переданных сервером"  
03  
04     /* действия с data */  
05  
06     function inner() {  
07         // ...  
08     }  
09  
10     return inner;  
11 }
```

```

08     }
09
10     data = null; // когда data станет не нужна -
11
12     return inner;
13 }

```

## jQuery: утечки и борьба с утечками

В jQuery для борьбы с утечками памяти в IE6-7 используется `$.data` API. Однако, это может стать причиной новых(!) утечек, характерных для jQuery.

Основной принцип `$.data` — это для любого JavaScript объекта сохранить/получить значение для элемента с помощью jQuery вызова:

```

1 $(document.body).data('prop', { anything: "любой объект"
  }) // set
2 alert( $(document.body).data('prop') ) // get

```

jQuery `elem.data(prop, val)` делает следующее:

1. Элемент получает уникальный идентификатор, если у него такого еще нет:

```
elem[ jQuery.expando ] = id = ++jQuery.uuid; // средствами jQuery
```

`jQuery.expando` — это случайная строка, сгенерированная jQuery один раз при входе на страницу. Уникальное свойство, чтобы ничего важного не перезаписать.

2. ...А сами данные сохраняются в специальном объекте `jQuery.cache`:

```
jQuery.cache[id]['prop'] = { anything: "любой объект" };
```

Когда данные считываются из элемента:

1. Уникальный идентификатор элемента извлекается из `id = elem[ jQuery.expando ]`.
2. Данные считываются из `jQuery.cache[id]`.

Смысл этого API в том, что DOM-элемент никогда не ссылается на JavaScript объект напрямую. Задействуется идентификатор, а сами данные хранятся в `jQuery.cache`. Утечек в IE не будет.

К тому же все данные известны библиотеке, так что можно клонировать с ними и т.п.

**Но как побочный эффект — утечка памяти, если элемент удален из DOM без дополнительной очистки.**

## Примеры утечек в jQuery

---

**Следующий код создает jQuery-утечку во всех браузерах:**

```
1 | $('<div/>')
2 |   .html(new Array(1000).join('text')) // div с текстом,
    возможна AJAX-загрузка
3 |   .click(function() { })
4 |   .appendTo('#data')
5 |
6 | document.getElementById('data').innerHTML = ''; // (*)
```

[Показать в отдельном окне](#)

Утечка происходит потому, что обработчик события в jQuery хранится в данных элемента. В строке (\*) элемент удален очисткой родительского `innerHTML`, но в `jQuery.cache` данные остались.

Более того, система обработки событий в jQuery устроена так, что вместе с обработчиком в данных хранится и ссылка на элемент, так что в итоге оба — и обработчик и элемент — остаются в памяти вместе со всем замыканием!

**Ещё более простой пример утечки:**

Этот код создает утечку:

```
1 | function go() {
2 |   $('<div/>')
3 |     .html(new Array(1000).join('text'))
4 |     .click(function() { })
5 | }
```

[Показать в отдельном окне](#)

Причина здесь в том, что элемент `<div>` создан, но нигде не размещен 😊. После выполнения функции ссылка на него теряется. Но обработчик события `click` уже сохранил данные в `jQuery.cache`, которые застревают там навсегда.

## Используем jQuery без утечек

---

**Чтобы избежать утечек, описанных выше, для удаления элементов используйте функции jQuery API, а не чистый JavaScript.**

Методы `remove()`, `empty()` и `html()` проверяют дочерние элементы на наличие данных и очищают их. Это несколько замедляет процедуру удаления, но зато освобождается память.

**К счастью обнаружить такие утечки легко. Проверьте размер `$.cache`. Если он большой и растёт, то изучите кэш, посмотрите, какие записи остаются и почему.**

## Улучшение производительности jQuery

---

У способа борьбы с утечками IE, применённого в jQuery, есть побочный эффект.

**Функции, удаляющие элементы, бегают по всему дереву DOM и очищают подэлементы.**

Представьте себе, что вы получили с сервера большую таблицу (в виде текста), вставили её в документ и хотите обновить. Вызов `$('table').remove()` будет бегать по всем ячейкам и искать в них данные. Но мы-то знаем, что обработчики назначены через делегирование, и тратить на это время ни к чему!

Чтобы «грязно» удалить элемент, без чистки, можно воспользоваться методом `detach()`. Его официальное назначение — в том, чтобы убрать элемент из DOM, но сохранить возможность для вставки (и, соответственно, оставить на нём все данные). А неофициальное — быстро убрать элемент из DOM. Если на нём нет данных и обработчиков, то всё хорошо.



В принципе, если хочется всё сделать чисто, но быстро — никто не мешает сделать `elem.detach()` и поместить вызов `elem.remove()` в `setTimeout`. В результате очистка будет происходить асинхронно и незаметно.

Итак, будем надеяться, что эта тема для вас теперь прозрачна и ясна и в следующих разделах мы не будем больше говорить об утечках в jQuery.

## Поиск и устранение утечек памяти

### Проверка на утечки

---

Существует множество шаблонов утечек и ошибок в браузерах, которые могут приводить к утечкам. Для их устранения сперва надо постараться изолировать и воспроизвести утечку.

- **Необходимо помнить, что браузер может очистить память не сразу когда объект стал недостижим, а чуть позже.** Например, сборщик мусора может ждать, пока не будет достигнут определенный лимит использования памяти, или запускаться время от времени.

Поэтому если вы думаете, что нашли проблему и тестовый код, запущенный в цикле, течёт — подождите примерно минуту, добейтесь, чтобы памяти ело стабильно и много. Тогда будет понятно, что это не особенность сборщика мусора.

- **Если речь об IE, то надо смотреть «Виртуальную память» в списке процессов, а не только обычную «Память».** Обычная может очищаться за счет того, что перемещается в виртуальную (на диск).
- Для простоты отладки, если есть подозрение на утечку конкретных объектов, в них добавляют большие свойства-маркеры. Например, подойдет фрагмент текста: `new Array(999999).join('leak')`.

### Настройка браузера

---

Утечки могут возникать из-за расширений браузера, взаимодействующих со страницей. Еще более важно, что **утечки могут быть следствием конфликта двух браузерных расширений** Например, было такое:

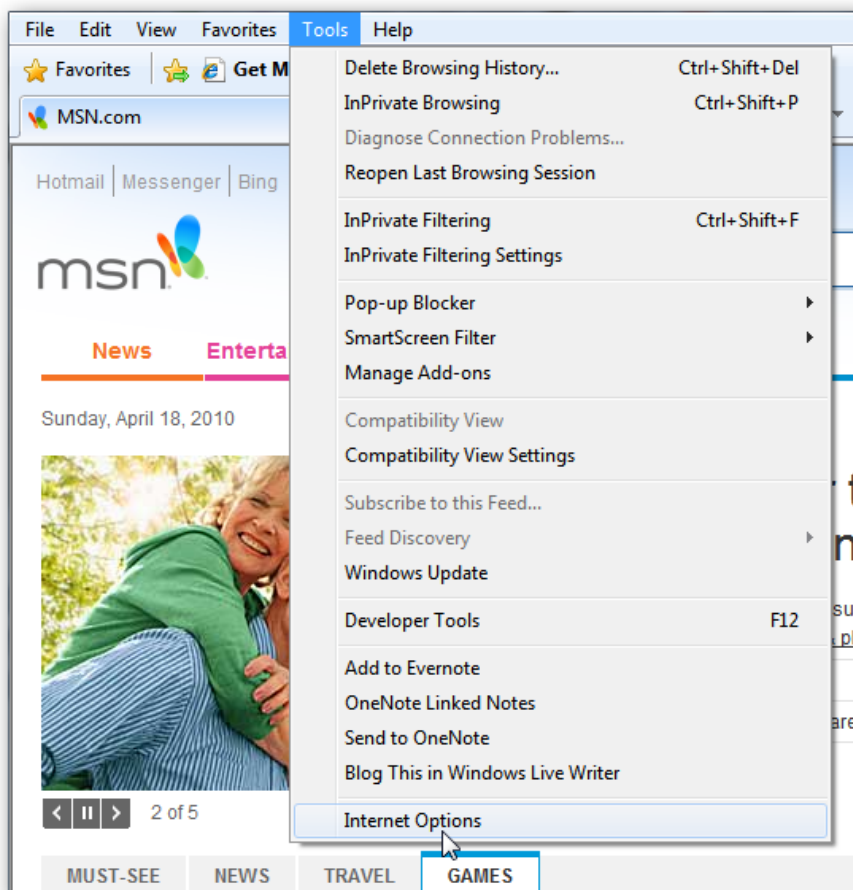
память текла когда включены расширения Skype и плагин антивируса одновременно.

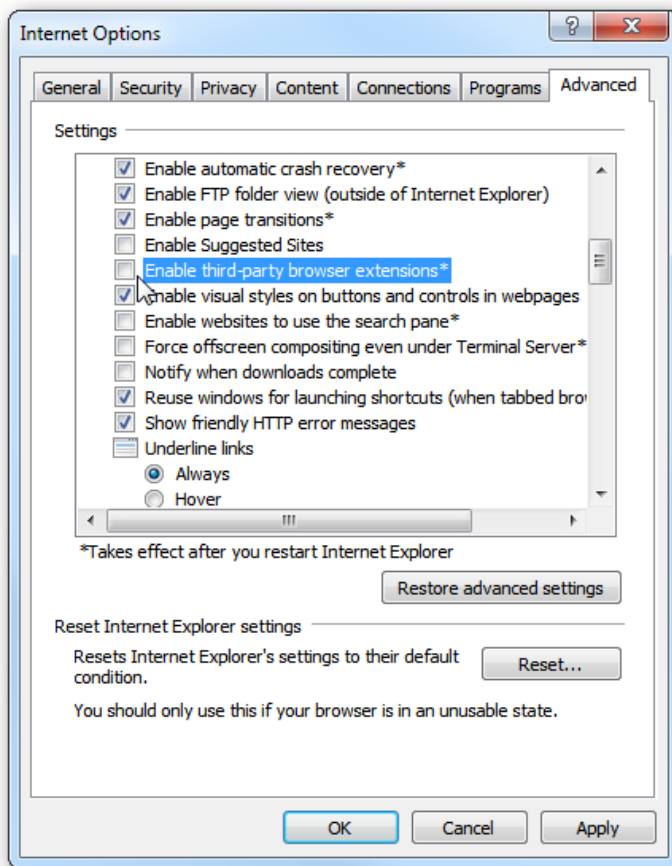
Чтобы понять, в расширениях дело или нет, нужно отключить их:

1. Отключить Flash.
2. Отключить антивирусную защиту, проверку ссылок и другие модули и дополнения.
3. Отключить плагины. Отключить VCE плагины.  
→ Для IE есть параметр командной строки:

"C:\Program Files\Internet Explorer\iexplore.exe" -extoff

Кроме того необходимо отключить сторонние расширения в свойствах IE.



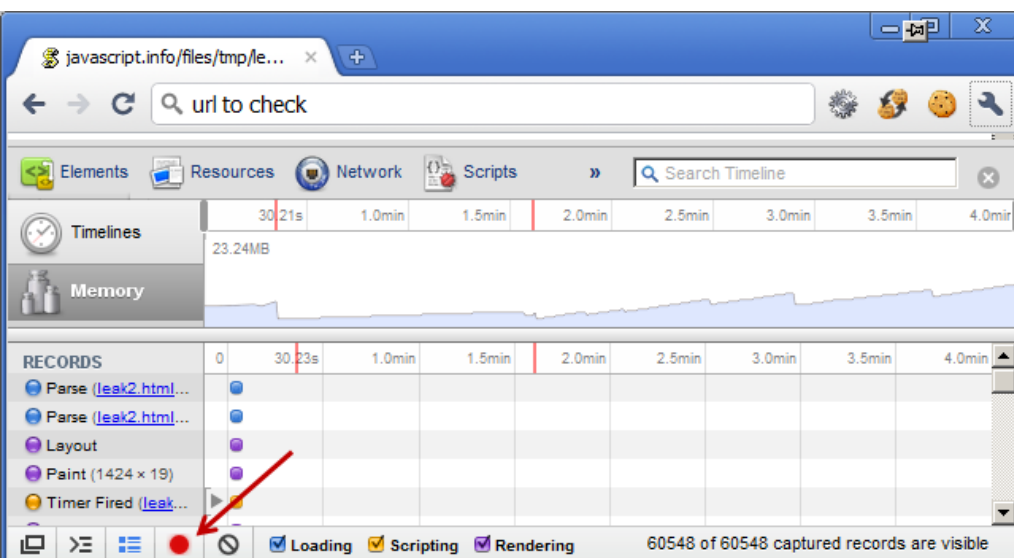


→ Firefox необходимо запускать с чистым профилем. Используйте следующую команду для запуска менеджера профилей и создания чистого пустого профиля:

```
firefox --profilemanager
```

## Инструменты

Пожалуй, единственный браузер с поддержкой отладки памяти — это Chrome. В инструментах разработчика вкладка Timeline - Memory показывает график использования памяти.



Можем посмотреть, сколько памяти и куда он использует.

Также в Profiles есть кнопка Take Heap Snapshot, здесь можно сделать и исследовать снимок текущего состояния страницы. Снимки можно сравнивать друг с другом, выяснять количество новых объектов. Можно смотреть, почему объект не очищен и кто на него ссылается.

Замечательная статья на эту тему есть в документации: [Chrome Developer Tools: Heap Profiling](#).

Утечки памяти штука довольно сложная. В борьбе с ними вам определенно понадобится одна вещь: *Удача!*



*Перевести с английского варианта  
учебника помог Марат Шагиев*

См. также

- [Chrome Developer Tools: Heap Profiling](#)
- [Memory leak when running setInterval in a new context \(Node.JS\)](#)
- [Circular Memory Leak Mitigation \(IE<8\)](#)
- [Understanding and Solving Internet Explorer Leak Patterns \(IE<8\)](#)
- [IE innerHTML Memory Leak \(IE<8\)](#)