

Современный учебник JavaScript

© Илья Кантор

Сборка от 27 апреля 2014 для печати

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.

Вопросы по сборке, предложения по её улучшению – можно писать мне, по адресу iliakan@javascript.ru .

Глава: Пишем на JavaScript

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

Содержание

Строгий режим, "use strict"

- "use strict"

- На уровне функции

- Использовать ли use strict?

Стиль кода

- Синтаксис

- Фигурные скобки

- Длина строки

- Отступы

- Точка с запятой

- Именованное

- Уровни вложенности

- Функции = Комментарии

- Функции — под кодом

- Комментарии

- Руководства по стилю

- Автоматизированные средства проверки

- Итого

Как писать неподдерживаемый код?

- Соглашения

- Краткость — сестра таланта!

- Именованное

- Однобуквенные переменные

- Не используйте i для цикла

- Русские слова и сокращения

- Будьте абстрактны при выборе имени

Похожие имена
А.К.Р.О.Н.И.М
Хитрые синонимы
Словарь терминов — это еда!
Повторно используйте имена
Добавляйте подчеркивания
Покажите вашу любовь к разработке
Перекрывайте внешние переменные
Мощные функции!
Внимание.. Сюр-при-из!
Заключение

Отладка в браузере Chrome

Общий вид панели Sources
Точки остановки
Остановиться и осмотреться
Управление выполнением
Консоль
Ошибки
Итого

Решения задач

Строгий режим, "use strict"

Современная спецификация языка содержит ряд несовместимых изменений, по сравнению со старым стандартом.

Чтобы не ломать существующий код, они, в основном, включаются при наличии специальной директивы `use strict`. Эта директива не поддерживается IE9-.

"use strict"

Директива выглядит как строка `"use strict"`; или `'use strict'`; , и может стоять в начале скрипта, либо в начале функции, например:

```
1 "use strict";  
2  
3 // этот код будет работать по стандарту ES5  
4 ...
```

Например, присвоение переменной без объявления в [старом стандарте \[1\]](#) было допустимо, а в современном — нет.

Поэтому следующий код выдаст ошибку:

```
1 "use strict";  
2  
3 x = 5; // error: x is not defined
```

Директиву нужно указывать до кода, иначе она не сработает:

```
1 var a;  
2  
3 "use strict";  
4  
5 x = 5; // ошибки не будет, строгий режим не включен
```

На уровне функции

Допустим, код работает только в старом режиме, но мы хотим написать новые функции, используя современный стандарт.

Директиву "use strict"; можно указать в начале функции, тогда она будет действовать только в ней.

Например, в коде ниже используются переменные без объявления. Но ошибка будет выведена только при запуске функции, так как именно она работает в строгом режиме:

```
01 function sayHi(person) {  
02   "use strict";  
03  
04   message = "Привет, " + person; // error: message is not defined  
05   //...  
06 }  
07  
08 person = "Вася";  
09  
10 sayHi(person);
```

Отменить действие "use strict" никак нельзя.

Если директива указана на уровне скрипта, то действует и на все функции.

Использовать ли use strict?

Строгий режим даёт две вещи:

1. Там, где в старом стандарте был «кривой код», в новом будет ошибка.
2. Некоторые возможности языка работают по-другому. Более корректно, но по-другому. По ходу учебника мы увидим много конкретных примеров этих различий.

Всё это хорошо.

Но основная проблема при использовании этой директивы — поддержка браузеров IE9-, которые игнорируют use strict и работают только в старом стандарте.

Действительно, предположим, что мы, используя эту директиву, разработали код в Chrome. Всё работает... Однако, вероятность ошибок при этом в IE9- выросла! Так как там поддерживается только старый стандарт, возможны ошибки совместимости. А отлаживать код, чтобы найти эти ошибки, в IE9- намного менее приятно, чем в Chrome.

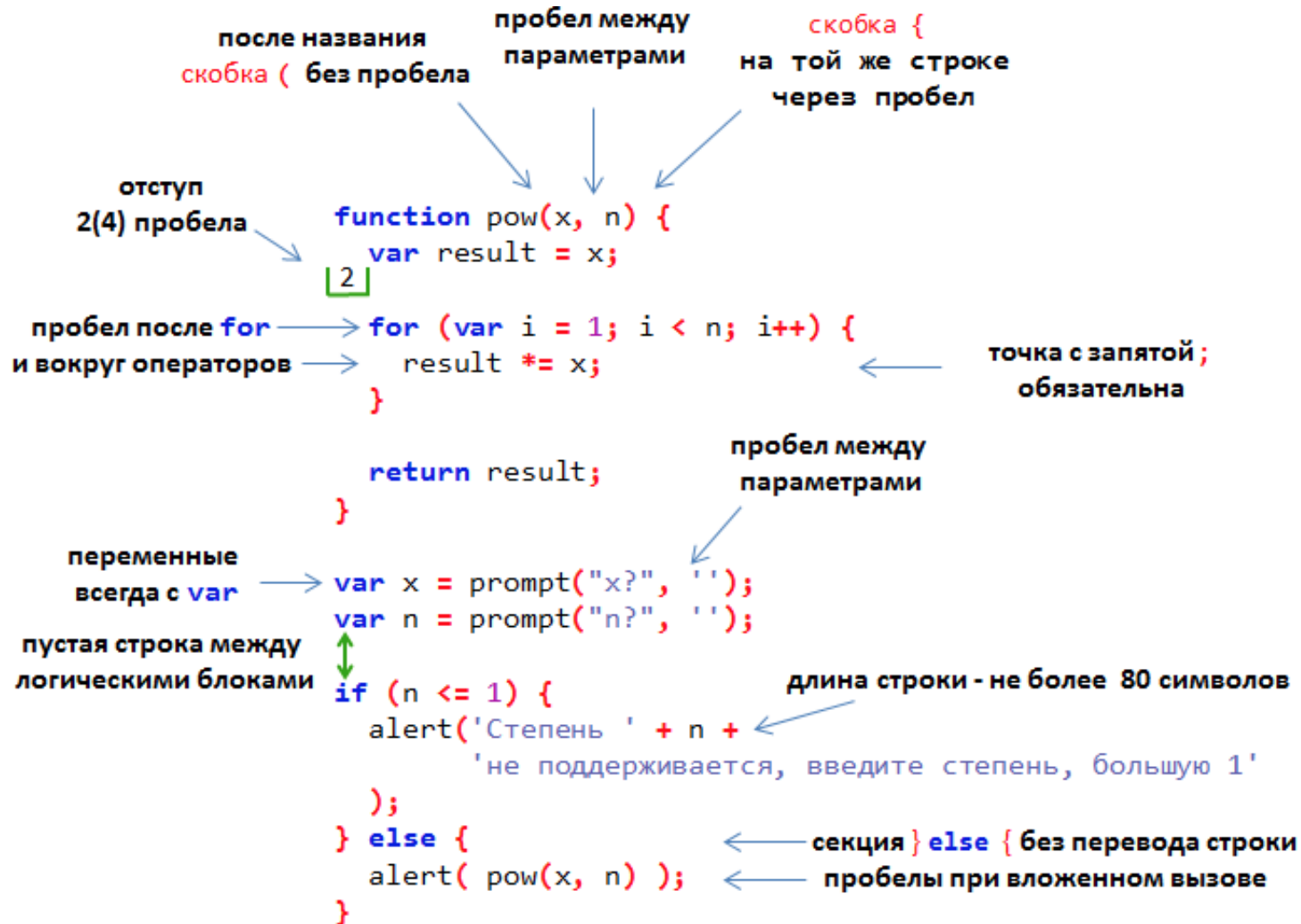
Тем не менее, строгий режим — это наше будущее. Поэтому его лучше использовать. Но при этом нужно очень хорошо знать отличия в работе JavaScript между старым и новым режимом, чтобы код, который вы напишете, не «ломался» при запуске в режиме старого стандарта.

Стиль кода

Код должен быть максимально читаемым и понятным. Для этого нужен *хороший стиль* написания кода. В этой главе мы рассмотрим компоненты такого стиля.

Синтаксис

Шпаргалка с правилами синтаксиса:



Разберём основные моменты.

Фигурные скобки

Пишутся на той же строке, так называемый «египетский» стиль. Перед скобкой — пробел.

Плохо! Скобки { } не нужны.

```
if (n <= 1) {alert('Степень ' + n + 'не поддерживается');}
```

В одну строку - **приемлемо**, если строка короткая

```
if (n <= 1) alert('Степень ' + n + 'не поддерживается');
```

Хорошо!

```
if (n <= 1) {  
    alert('Степень ' + n + 'не поддерживается');  
}
```

Если у вас уже есть опыт в разработке и вы привыкли делать скобку на отдельной строке — это тоже вариант. В конце концов, решать вам. Но в основных JavaScript-фреймворках (jQuery, Dojo, Google Closure Library, Mootools, ExtJS, YUI...) стиль именно такой.

Если условие и код достаточно короткие, например `if (cond) return null;`, то запись в одну строку вполне читаема... Но, как правило, отдельная строка всё равно воспринимается лучше.

Длина строки

Максимальную длину строки согласовывают в команде. Как правило, это либо 80, либо 120 символов, в зависимости от того, какие мониторы у разработчиков.

Более длинные строки необходимо разбивать. Если этого не сделать, то перевод очень длинной строки сделает редактор, и это может быть менее красиво и читаемо.

Отступы

Отступы нужны двух типов:

➡ **Горизонтальный отступ, при вложенности — два(или четыре) пробела.**

Как правило, используются именно пробелы, т.к. они позволяют сделать более гибкие «конфигурации отступов», чем символ «Tab».

Например:

```

1 function removeClass(obj, cls) {
2   var targetObj = obj,
3     targetClass = cls,    // <-- пробельный отступ выровнен!
4     className = obj.className; // <--
5   ...
6 }

```

Переменные здесь объявлены по вертикали, т.к. вообще человеческий глаз лучше воспринимает («сканирует») вертикально выравненную информацию, нежели по горизонтали. Это известный факт среди дизайнеров и нам, программистам, он тоже будет полезен для лучшей организации кода.

➔ Вертикальный отступ, для лучшей разбивки кода — перевод строки.

Используется, чтобы разделить логические блоки внутри одной функции. В примере ниже разделены функция `row`, получение данных `x`, `n` и их обработка `if`.

```

01 function row(..) {
02   ..
03 }
04           // <--
05 x = ...
06 n = ...
07           // <--
08 if (n <= 1) {
09   ...
10 }
11 ..

```

Вставляйте дополнительный перевод строки туда, где это сделает код более читаемым. Не должно быть более 9 строк кода подряд без вертикального отступа.

Точка с запятой

Точки с запятой нужно ставить, даже если их, казалось бы, можно пропустить.

Есть языки, в которых точка с запятой не обязательна, и её там никто не ставит. В JavaScript она тоже не обязательна, но ставить нужно. В чём же разница?

Она в том, что **в JavaScript без точки с запятой возможны трудноуловимые ошибки**. С некоторыми примерами вы встретитесь дальше в учебнике. Такая вот особенность синтаксиса. И поэтому рекомендуется её всегда ставить.

Именованние

Общее правило:

- ➔ Имя переменной — существительное.
- ➔ Имя функции — глагол или начинается с глагола. Бывает, что имена для краткости делают существительными, но глаголы понятнее.

Для имён используется английский язык (не транслит) и верблюжья нотация.

Более подробно — читайте про [имена функций \[2\]](#) и [имена переменных \[3\]](#).

Уровни вложенности

Уровней вложенности должно быть немного.

Например, проверки в циклах лучше делать через «continue» [4], чтобы не было дополнительного уровня `if(..) { ... }`:

Вместо:

```
1 for (var i=0; i<10; i++) {
2   if (i подходит) {
3     ... // <- уровень вложенности 2
4   }
5 }
```

Используйте:

```
1 for (var i=0; i<10; i++) {
2   if (i не подходит) continue;
3   ... // <- уровень вложенности 1
4 }
```

Аналогичная ситуация — с `if/else` и `return`. Следующие две конструкции идентичны.

Первая:

```
1 function isEven(n) { // проверка чётности
2   if (n % 2 == 0) {
3     return true;
4   } else {
5     return false;
6   }
7 }
```

Вторая:

```
1 function isEven(n) { // проверка чётности
2   if (n % 2 == 0) {
3     return true;
4   }
5
6   return false;
7 }
```

Если в блоке `if` идёт `return`, то `else` за ним не нужен.

Лучше быстро обработать простые случаи, вернуть результат, а дальше разбираться со сложным, без дополнительного уровня вложенности.

В случае с функцией `isEven` можно было бы поступить и проще:

```
function isEven(n) { // проверка чётности
  return n % 2 == 0;
}
```

..Казалось бы, можно пойти дальше, есть ещё более короткий вариант:

```
function isEven(n) { // проверка чётности
    return !(n % 2);
}
```

...Однако, код `!(n % 2)` менее очевиден чем `n % 2 == 0`. Поэтому, на самом деле, последний вариант хуже. **Главное для нас — не краткость кода, а его простота и читаемость.**

Функции = Комментарии

Функции должны быть небольшими. Если функция большая — желательно разбить её на несколько.

Этому правилу бывает сложно следовать, но оно стоит того. При чем же здесь комментарии?

Вызов отдельной небольшой функции не только легче отлаживать и тестировать — сам факт его наличия является *отличным комментарием*.

Сравните, например, две функции `showPrimes(n)` для вывода простых чисел до `n`.

Первый вариант:

```
01 function showPrimes(n) {
02     nextPrime:
03     for (var i=2; i<n; i++) {
04
05         for (var j=2; j<i; j++) {
06             if ( i % j == 0) continue nextPrime;
07         }
08
09         alert(i); // простое
10     }
11 }
```

Второй вариант, вынесена подфункция `isPrime(n)` для проверки на простоту:

```
01 function showPrimes(n) {
02
03     for (var i=2; i<n; i++) {
04         if (!isPrime(i)) continue;
05
06         alert(i); // простое
07     }
08 }
09
10 function isPrime(n) {
11     for (var i=2; i<n; i++) {
12         if ( n % i == 0) return false;
13     }
14     return true;
15 }
```

Второй вариант проще и понятнее, не правда ли? Вместо участка кода мы видим описание действия, которое там совершается (проверка `isPrime`).

Функции — под кодом

Есть два способа расположить функции, необходимые для выполнения кода.

1. Функции над кодом, который их использует:

```
01 // объявить функции
02 function createElement() {
03     ...
04 }
05
06 function setHandler(elem) {
07     ...
08 }
09
10 function walkAround() {
11     ...
12 }
13
14 // код, использующий функции
15 var elem = createElement();
16 setHandler(elem);
17 walkAround();
```

2. Сначала код, а функции внизу:

```
01 // код, использующий функции
02 var elem = createElement();
03 setHandler(elem);
04 walkAround();
05
06 // --- функции ---
07
08 function createElement() {
09     ...
10 }
11
12 function setHandler(elem) {
13     ...
14 }
15
16 function walkAround() {
17     ...
18 }
```

...На самом деле существует еще третий «стиль», при котором функции хаотично разбросаны по коду 😊, но это ведь не наш метод, да?

Как правило, лучше располагать функции под кодом, который их использует. То есть, это 2й способ.

Дело в том, что при чтении такого кода мы хотим знать в первую очередь, *что он делает*, а уже затем *какие функции ему помогают*. Если первым идёт код, то это как раз дает необходимую информацию. Что же касается функций, то вполне возможно нам и не понадобится их читать, особенно если они названы адекватно и то, что они делают, понятно.

У первого способа, впрочем, есть то преимущество, что на момент чтения мы уже знаем, какие функции существуют.

Таким образом, если над названиями функций никто не думает — может быть, это будет лучшим выбором 😊. Попробуйте оба варианта, но по моей практике предпочтителен всё же второй.

Комментарии

В коде нужны комментарии. Их можно условно разделить на несколько типов.

1. Справочный комментарий перед функцией — о том, что именно она делает, какие параметры принимает и что возвращает.

Для таких комментариев существует синтаксис [JSDoc \[5\]](#) .

```
01  /**  
02   * Возвращает x в степени n, только для натуральных n  
03   *  
04   * @param {number} x Число для возведения в степень.  
05   * @param {number} n Показатель степени, натуральное число.  
06   * @return {number} x в степени n.  
07   */  
08  function pow(x, n) {  
09    ...  
10  }
```

Такие комментарии обрабатываются многими редакторами, например [Aptana \[6\]](#) и редакторами от [JetBrains \[7\]](#). Они учитывают их при автодополнении.

2. Краткий комментарий, что именно происходит в данном блоке кода.

В хорошем коде он нужен редко, так как всё понятно из переменных, имён функций.

3. Есть несколько способов решения задачи. Почему выбран именно этот?

Как правило, из кода можно понять, что он делает. Бывает, конечно, всякое, но, в конце концов, вы этот код видите. Гораздо важнее может быть то, чего вы *не видите!*

Почему это сделано именно так? На это сам код ответа не даёт.

Например, пробовали решить задачу по-другому, но не получилось — напишите об этом. Почему вы выбрали именно этот способ решения? Особенно это важно в тех случаях, когда используется не первый приходящий в голову способ, а какой-то другой.

Без этого возможна, например, такая ситуация:

- ➡ Вы открываете код, который был написан какое-то время назад, и видите, что он «неоптимален».
- ➡ Думаете: «Какой я был дурак», и переписываете под «более очевидный и правильный» вариант.
- ➡ ...Порыв, конечно, хороший, да только этот вариант вы уже обдумали раньше. И отказались, а почему — забыли. В процессе переписывания вспомнили, конечно (к счастью), но результат - потеря времени на повторное обдумывание.

Комментарии, которые объясняют поведение кода, очень важны. Они помогают понять происходящее и принять правильное решение о развитии кода.

4. Какие неочевидные возможности обеспечивает этот код? Где в другом месте кода они используются?

В хорошем коде должно быть минимум неочевидного. Но там, где это есть — пожалуйста, комментируйте.

5. Архитектурный комментарий — «как оно, вообще, устроено». Применённые технологии, поток взаимодействия. Для этого, в частности, используется [UML \[8\]](#) , но можно и без него. Главное — чтобы понятно.

Что интересно, в коде начинающих разработчиков большинство комментариев обычно типа 2. Но на самом деле именно эти комментарии являются самыми ненужными. А всё дело в том, что людям лень придумывать правильные имена и структурировать функции. Ну ничего. Жизнь заставит :/

Руководства по стилю

Когда написанием проекта занимается целая команда, то должен существовать один стандарт кода, описывающий где и когда ставить пробелы, запятые, переносы строк и т.п.

Сейчас, когда есть столько готовых проектов, нет смысла придумывать целиком своё руководство по стилю. Можно взять уже готовое, и которому, по желанию, всегда можно что-то добавить.

Большинство есть на английском, сообщите мне, если найдёте хороший перевод:

- ➡ [Google JavaScript Style Guide \[9\]](#)
- ➡ [jQuery Core Style Guidelines \[10\]](#)
- ➡ [Idiomatic.JS \[11\]](#) (есть [перевод \[12\]](#))
- ➡ [Dojo Style Guide \[13\]](#)

Для того, чтобы начать разработку, вполне хватит элементов стилей, обозначенных в этой главе. В дальнейшем, посмотрите на эти руководства, найдите «свой» стиль 😊

Автоматизированные средства проверки

Существуют онлайн-сервисы, проверяющие стиль кода.

Самые известные — это:

- ➡ [JSLint \[14\]](#) — проверяет код на соответствие [стилю JSLint \[15\]](#), в онлайн-интерфейсе вверху можно ввести код, а внизу различные настройки проверки, чтобы сделать её более мягкой.
- ➡ [JSHint \[16\]](#) — ещё один вариант JSLint, ослабляющий требования в ряде мест.
- ➡ [Closure Linter \[17\]](#) — проверка на соответствие [Google JavaScript Style Guide \[18\]](#).

Все они также доступны в виде программ, которые можно скачать.

Итого

Описанные принципы оформления кода уместны в большинстве проектов. Есть и другие полезные соглашения.

Следуя (или не следуя) им, необходимо помнить, что любые советы по стилю хороши лишь тогда, когда делают код читаемее, понятнее, проще в поддержке.

Как писать неподдерживаемый код?

Предлагаю вашему вниманию советы мастеров древности, следование которым создаст дополнительные рабочие места для JavaScript-разработчиков.

Код, который вы напишете, будет так сложен в поддержке, что у JavaScript'еров, которые придут после вас, даже простейшее изменение займет годы оплачиваемого труда!

Более того, *внимательно* следуя этим правилам, вы сохраните и своё рабочее место, так как все будут бояться вашего кода и бежать от него...

...Впрочем, всему своя мера. Код не должен **выглядеть** сложным в поддержке — подобное напишет любой дурак. Код должен **быть**

таковым. Иначе это заметят, и код будет переписан с нуля. Вы не можете такого допустить. Эти советы учитывают такую возможность. Да здравствует дзен.

Статья представляет собой вольный перевод [How To Write Unmaintainable Code \[19\]](#) с дополнениями для JavaScript.

Соглашения

*Рабочий-чистильщик: "...Вот только жук у вас необычный...
И чтобы с ним справиться, я должен жить как жук, стать жуком, думать как жук."
(грызёт стол Симпсонов)
Сериял "Симпсоны", серия Helter Shelter*

Чтобы помешать другому программисту исправить ваш код, вы должны понять путь его мыслей.

Представьте, перед ним — ваш большой скрипт. И ему нужно поправить его. У него нет ни времени ни желания, чтобы читать его целиком, а тем более — досконально разбирать. Он хотел бы по-быстрому найти нужное место, сделать изменение и убраться восвояси без появления побочных эффектов.

Он рассматривает ваш код как бы через трубочку из туалетной бумаги. Это не даёт ему общей картины, он ищет тот небольшой фрагмент, который ему необходимо изменить. По крайней мере, он надеется, что этот фрагмент будет небольшим.

На что он попытается опереться в этом поиске — так это на соглашения, принятые в программировании, об именах переменных, названиях функций и методов...

Как затруднить задачу? Можно везде нарушать соглашения — это мешает ему, но такое могут заметить, и код будет переписан. Как поступил бы ниндзя на вашем месте?

...Правильно! Следуйте соглашениям «в общем», но иногда — нарушайте их. Тщательно разбросанные по коду нарушения соглашений с одной стороны не делают код явно плохим при первом взгляде, а с другой — имеют в точности тот же, и даже лучший эффект, чем явное неследование им!

Например, в jQuery есть метод [wrap \[20\]](#), который обортывает один элемент вокруг другого:

```
1 var img = $('<img/>');
2 var div = $('<div/>');
3
4 img.wrap(div); // обернуть img в div
```

Результат кода выше:

```
<div>
  <img/>
</div>
```

(div обернулся вокруг img)

А теперь, когда все расслабились и насладились этим замечательным методом...

...Самое время ниндзя нанести свой удар!

Как вы думаете, что будет, если добавить к коду выше строку:

```
5 div.append('<span/>');
```

Возможно, вы полагаете, что добавится в конец div, сразу после img?

А вот и нет! А вот и нет!..

Такое трудно себе представить, но вызов `img.wrap(div)` внутри клонирует `div`. И оборачивает вокруг `img` не сам `div`, а его ~~этой~~ клон.

При этом исходный `div` остаётся пустым. После применения к нему `append` получается два `div` 'а: один обернут вокруг `span`, а в другом — только `img`.

Переменная <code>div</code>	Клон <code>div</code> , созданный <code>wrap</code> (не присвоен никакой переменной)
<pre><div> </div></pre>	<pre><div> </div></pre>

Соглашение в данном случае — в том, что большинство методов jQuery не копируют элементы. А вызов `wrap` — копирует.

Код его истинный ниндзя писал!

Краткость — сестра таланта!

Пишите «как короче», а не как понятнее. «Меньше букв» — уважительная причина для нарушения любых соглашений.

Ваш верный помощник — возможности языка, использованные неочевидным образом.

Обратите внимание на оператор вопросительный знак '?', например:

```
// код из jQuery
i = i ? i < 0 ? Math.max( 0, len + i ) : i : 0;
```

Разработчик, встретивший эту строку и попытавшийся понять, чему же всё-таки равно `i`, скорее всего придёт к вам за разъяснениями. Смело скажите ему, что короче — это всегда лучше. Посвятите и его в пути ниндзя. Не забудьте вручить [Дао дэ цзин \[21\]](#).

Именованное

Существенную часть науки о создании неподдерживаемого кода занимает искусство выбора имён.

Однобуквенные переменные

Называйте переменные коротко: `a`, `b` или `c`.

В этом случае никто не сможет найти её, используя функцию «Поиск» текстового редактора.

Более того, даже найдя — никто не сможет «расшифровать» её и догадаться, что она означает.

Не используйте `i` для цикла

В тех местах, где однобуквенные переменные общеприняты, например, в счетчике цикла — ни в коем случае не используйте стандартные названия `i`, `j`, `k`. Где угодно, только не здесь!

Остановите свой взыскательный взгляд на чём-нибудь более экзотическом. Например, `x` или `y`.

Эффективность этого подхода особенно заметна, если тело цикла занимает одну-две страницы.

В этом случае заметить, что переменная — счетчик цикла, без пролистывания вверх, невозможно.

Русские слова и сокращения

Если вам *приходится* использовать длинные, понятные имена переменных — что поделаться. Но и здесь есть простор для творчества!

Назовите переменные «калькой» с русского языка или как-то «улучшите» английское слово.

В одном месте напишите `var ssilka`, в другом `var ssylka`, в третьем `var link`, в четвёртом — `var lnk...` Это действительно великолепно работает и очень креативно!

Количество ошибок при поддержке такого кода увеличивается во много раз.

Будьте абстрактны при выборе имени

*Лучший кувшин лепят всю жизнь.
Высокая музыка неподвластна слуху.
Великий образ не имеет формы.
Лао-цзы*

При выборе имени старайтесь применить максимально абстрактное слово, например `obj`, `data`, `value`, `item`, `elem` и т.п.

➡ **Идеальное имя для переменной: `data`.** Используйте это имя везде, где можно. В конце концов, каждая переменная содержит *данные*, не правда ли?

Но что делать, если имя `data` уже занято? Попробуйте `value`, оно не менее универсально. Ведь каждая переменная содержит *значение*.

Занято и это? Есть и другой вариант.

➡ **Называйте переменную по типу: `obj`, `num`, `arr...`**

Насколько это усложнит разработку? Как ни странно, намного!

Казалось бы, название переменной содержит информацию, говорит о том, что в переменной — число, объект или массив... С другой стороны, **когда непосвящённый будет разбирать этот код — он с удивлением обнаружит, что информации нет!**

Ведь как раз тип легко понять, запустив отладчик и посмотрев, что внутри. Но в чём смысл этой переменной? Что за массив/объект/число в ней хранится? Без долгой медитации над кодом тут не обойтись!

➡ **Что делать, если и эти имена кончились? Просто добавьте цифру: `item1`, `item2`, `elem5`, `data1..`**

Похожие имена

Только истинно внимательный программист достоин понять ваш код. Но как проверить, достоин ли читающий?

Один из способов — использовать похожие имена переменных, например `data` и `date`. Бегло прочитав такой код почти невозможно. А уж заметить опечатку и поправить её... Ммммм... Мы здесь надолго, время попить чайку.

А.К.Р.О.Н.И.М

Используйте сокращения, чтобы сделать код короче. Например `ie` (Inner Element), `mc` (Money Counter) и другие. Если вы обнаружите, что путаетесь в них сами — героически страдайте, но не переписывайте код. Вы знали, на что шли.

Очень трудно найти чёрную кошку в тёмной комнате, особенно когда её там нет.
Конфуций

Чтобы было не скучно — используйте *похожие названия* для обозначения *одинаковых действий*.

Например, если метод показывает что-то на экране — начните его название с `display..` (скажем, `displayElement`), а в другом месте объявите аналогичный метод как `show..` (`showFrame`).

Как бы намекайте этим, что существует тонкое различие между способами показа в этих методах, хотя на самом деле его нет.

По возможности, договоритесь с членами своей команды. Если Вася в своих классах использует `display..`, то Валера — обязательно `render..`, а Петя — `paint...`

...И напротив, если есть две функции с важными отличиями — используйте одно и то же слово для их описания! Например, с `print...` можно начать метод печати на принтере `printPage`, а также — метод добавления текста на страницу `printText`.

А теперь, пусть читающий код думает: «Куда же выводит сообщение `printMessage`?». Особый шик — добавить элемент неожиданности. Пусть `printMessage` выводит не туда, куда все, а в новое окно.

Словарь терминов — это еда!

Ни в коем случае не поддавайтесь требованиям написать словарь терминов для проекта. Если же он уже есть — не следуйте ему, а лучше проглотите и скажите, что так и былО!

Пусть читающий ваш код программист напрасно ищет различия в `helloUser` и `welcomeVisitor` и пытается понять, когда что использовать. Вы-то знаете, что на самом деле различий нет, но искать их можно о-очень долго.

Для обозначения посетителя в одном месте используйте `user`, а в другом `visitor`, в третьем — просто `u`. Выбирайте одно имя или другое, в зависимости от функции и настроения.

Это воплотит сразу два ключевых принципа ниндзя-дизайна — *сокрытие информации* и *подмена понятий*!

Повторно используйте имена

По возможности, повторно используйте имена переменных, функций и свойств. Просто записывайте в них новые значения.

Добавляйте новое имя только если это абсолютно необходимо.

В функции старайтесь обойтись только теми переменными, которые были переданы как параметры.

Это не только затруднит идентификацию того, что *сейчас* находится в переменной, но и сделает почти невозможным поиск места, в котором конкретное значение было присвоено.

Цель — максимально усложнить отладку и заставить читающего код программиста построчно анализировать код и конспектировать изменения переменных для каждой ветки исполнения.

Продвинутый вариант этого подхода — незаметно (!) подменить переменную на нечто похожее, например:

```

1 function ninjaFunction(elem) {
2   // 20 строк кода, работающего с elem
3
4   elem = elem.cloneNode(true);
5
6   // еще 20 строк кода, работающего с elem
7 }

```

Программист, пожелавший добавить действия с `elem` во вторую часть функции, будет удивлён. Лишь во время отладки, посмотрев весь код, он с удивлением обнаружит, что оказывается имел дело с клоном!

Регулярные встречи с этим приемом на практике говорят: защититься невозможно. Эффективно даже против опытного ниндзи.

Добавляйте подчеркивания

Добавляйте подчеркивания `_` и `__` к именам переменных. Желательно, чтобы их смысл был известен только вам, а лучше — вообще без явной причины.

Этим вы достигните двух целей. Во-первых, код станет длиннее и менее читаемым, а во-вторых, другой программист будет долго искать смысл в подчёркиваниях. Особенно хорошо сработает и внесет сумятицу в его мысли, если в некоторых частях проекта подчеркивания будут, а в некоторых — нет.

В процессе развития кода вы, скорее всего, будете путаться и смешивать стили: добавлять имена с подчеркиваниями там, где обычно подчеркиваний нет, и наоборот. Это нормально и полностью соответствует третьей цели — увеличить количество ошибок при внесении исправлений.

Покажите вашу любовь к разработке

Пусть все видят, какими замечательными сущностями вы оперируете! Имена `superElement`, `megaFrame` и `niceItem` при благоприятном положении звёзд могут привести к просветлению читающего.

Действительно, с одной стороны, кое-что написано: `super...`, `mega...`, `nice...`. С другой — это не несёт никакой конкретики. Читающий может решить поискать в этом глубинный смысл и замедлитесь на часок-другой оплаченного рабочего времени.

Перекрывайте внешние переменные

*Находясь на свету, нельзя ничего увидеть в темноте.
Пребывая же в темноте, увидишь все, что находится на свету.
Гуань Инь-цзы*

Почему бы не использовать одинаковые переменные внутри и снаружи функции? Это просто и не требует придумывать новых имён.

```

1 var user = authenticateUser();
2
3 function render() {
4   var user = ...
5   ...
6   ...
7   ... // <-- программист захочет внести исправления сюда, и..
8   ...
9 }

```

Зашедший в середину метода `render` программист, скорее всего, не заметит, что переменная `user` «уже не та» и использует её... Ловушка захлопнулась! Здравствуй, отладчик.

Мощные функции!

Не ограничивайте действия функции тем, что написано в её названии. Будьте шире.

Например, функция `validateEmail(email)` может, кроме проверки e-mail на правильность, выводить сообщение об ошибке и просить заново ввести e-mail.

Выберите хотя бы пару дополнительных действий, кроме основного назначения функции. Главное — они должны быть неочевидны из названия функции. Истинный ниндзя-девелопер сделает так, что они будут неочевидны и из кода тоже.

Объединение нескольких смежных действий в одну функцию защитит ваш код от повторного использования.

Представьте, что другому разработчику нужно только проверить адрес, а сообщение — не выводить. Ваша функция `validateEmail(email)`, которая делает и то и другое, ему не подойдёт. Работодатель будет вынужден оплатить создание новой.

Внимание.. Сюр-при-из!

Есть функции, название которых говорит о том, что они ничего не меняют. Например, `isReady`, `checkPermission`, `findTags`.. В трактатах это называется «отсутствие побочных эффектов».

По-настоящему красивый приём — делать в таких функциях что-нибудь полезное, заодно с процессом проверки. Что именно — совершенно неважно.

Удивление и ошеломление, которое возникнет у вашего коллеги, когда он увидит, что функция с названием на `is..`, `check..` или `find...` что-то меняет — несомненно, расширит его границы разумного!

Ещё одна вариация такого подхода — возвращать нестандартное значение.

Ведь общеизвестно, что `is..` и `check..` обычно возвращают `true/false`. Проявите оригинальное мышление. Пусть вызов `checkPermission` возвращает не результат `true/false`, а объект с результатами проверки! А что, полезно.

Те же, кто попытается написать проверку `if (checkPermission(..))`, будут удивлены результатом. Ответьте им: «надо читать документацию!». И перешлите эту статью.

Заключение

Все советы выше пришли из реального кода... И в том числе от разработчиков с большим опытом.

Возможно, даже больше вашего, так что не судите опрометчиво 😊

- ➡ Следуйте нескольким из них — и ваш код станет полон сюрпризов.
- ➡ Следуйте многим — и ваш код станет истинно вашим, никто не захочет изменять его.
- ➡ Следуйте всем — и ваш код станет ценным уроком для молодых разработчиков, ищущих просветления.

Отладка в браузере Chrome

Перед тем, как двигаться дальше, поговорим об отладке скриптов.

Все современные браузеры поддерживают для этого «инструменты разработчика». Исправление ошибок с их помощью намного проще и быстрее.

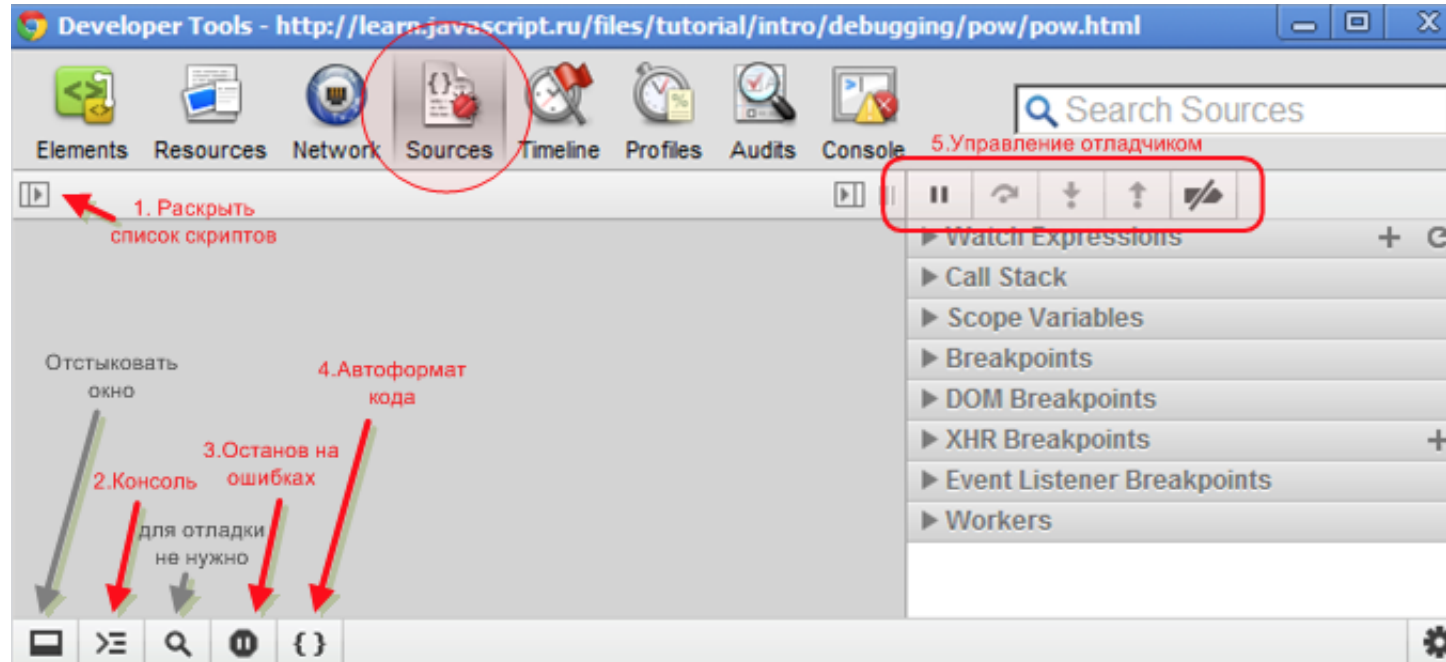
На текущий момент самые многофункциональные инструменты — в браузере Chrome. Также очень хорош Firebug (для Firefox).

Общий вид панели Sources

Зайдите на страницу [tutorial/debugging/pow/index.html](http://learn.javascript.ru/files/tutorial/intro/debugging/pow/pow.html) [22] браузером Chrome.

Откройте инструменты разработчика: F12 или в меню Инструменты > Инструменты Разработчика.

Выберите сверху Sources:



Кнопки, которые мы будем использовать:

1. Раскрывает список скриптов и стилей, подключённых к странице.
2. Включает-выключает консоль (ESC).
3. Двойное нажатие на эту кнопку заставляет отладчик останавливаться на ошибках JavaScript (кнопка поменяет цвет на фиолетовый).
4. Код можно отлаживать в отформатированном виде, если включить эту кнопку.
5. Панель управления потоком исполнения в режиме отладки. Нам она скоро понадобится.

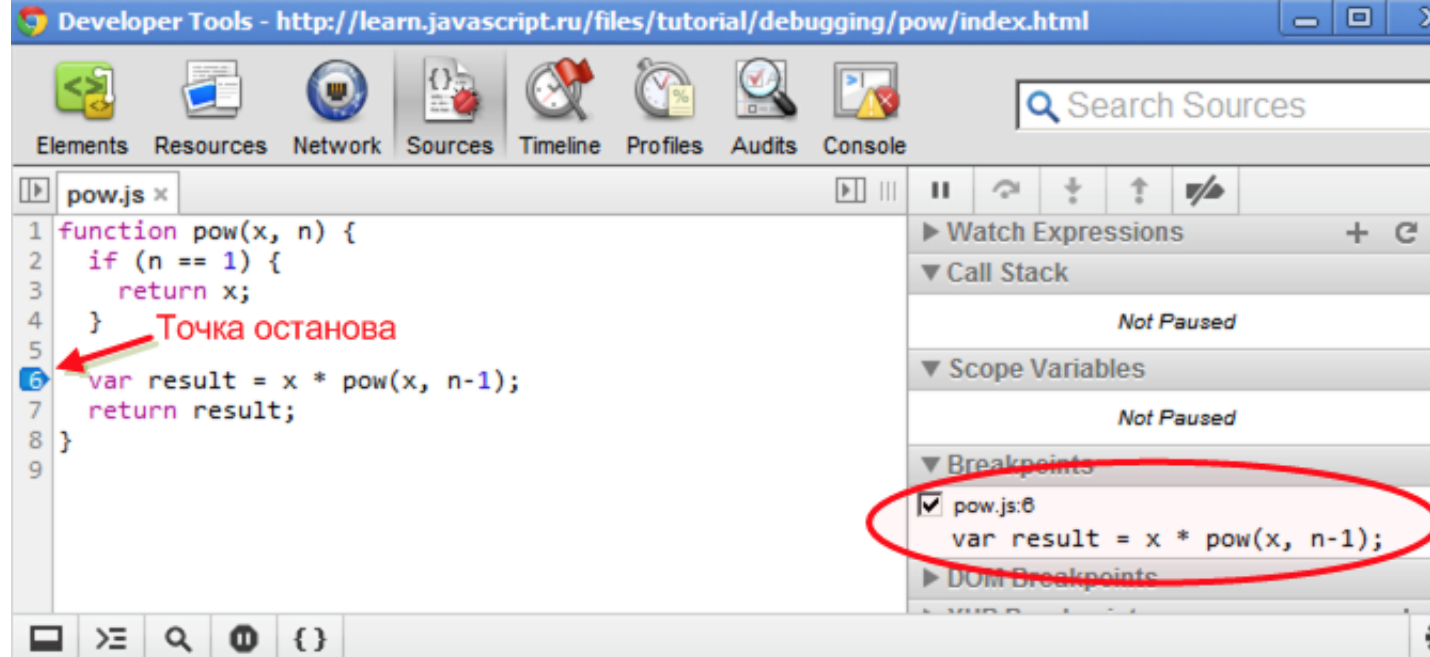
Точки останова

Зашли на страницу? Раскройте список скриптов, нажав на кнопку 1, и выберите `pow.js`. Кликните на 6й строке, прямо на цифре 6.

Поздравляю! Вы поставили свой первый «брейкпойнт».

Слово *Брейкпойнт* (breakpoint) — часто используемый английский жаргонизм. В русскоязычных пособиях используется термин «точка останова». Это то место в коде, где отладчик будет останавливать выполнение JavaScript, как только оно до него дойдёт.

В остановленном коде можно посмотреть любые переменные, выполнить команды и т.п.



Справа-снизу находится вкладка Breakpoints, в ней можно:

- ➡ Просматривать точки останова.
- ➡ Выключать их кликом на чекбокс.
- ➡ При клике на точку останова происходит переход на соответствующее место кода (удобно, когда их много).



Дополнительные возможности

- ➡ Остановку можно инициировать и напрямую из кода скрипта, командой `debugger`:

```
1 function pow(x, n) {  
2   ...  
3   debugger; // <-- отладчик остановится тут  
4   ...  
5 }
```

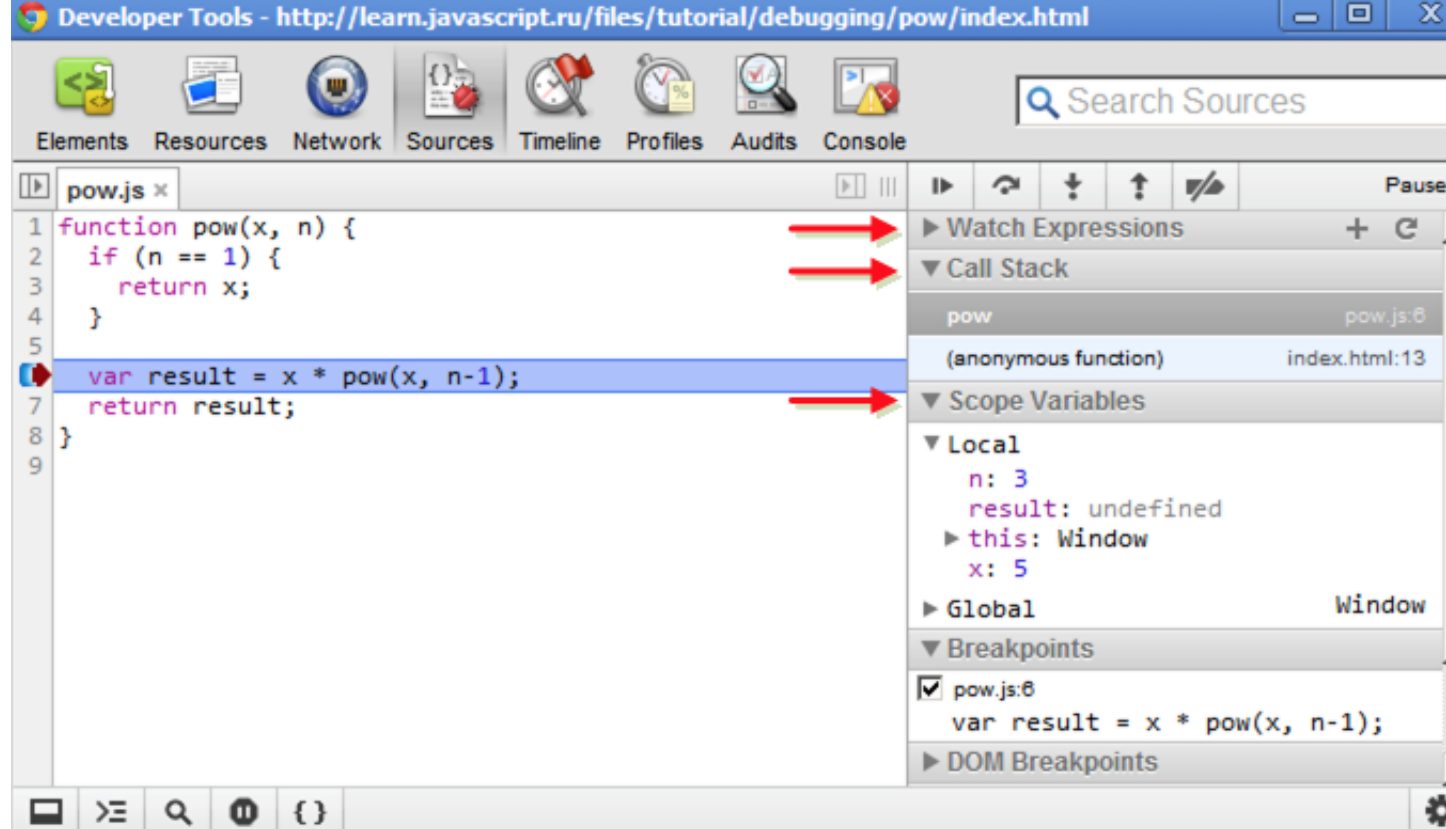
- ➡ Правый клик на цифру строки позволяет создать условную точку останова (conditional breakpoint), т.е. задать условие, при котором точка останова работает.

Это удобно, если остановка нужна только при определённом значении переменной или параметра функции.

Остановиться и осмотреться

Наша функция выполняется сразу при загрузке страницы, так что самый простой способ активировать JavaScript — перезагрузить её. Итак, нажимаем F5.

Если вы сделали всё, как описано выше, то выполнение прервётся как раз на 6й строке.



Обратите внимание на информационные вкладки справа (отмечены стрелками).

В них мы можем посмотреть текущее состояние:

1. **Watch Expressions** — показывает текущие значения любых выражений.

Можно раскрыть эту вкладку, нажать мышью + на ней и ввести любое выражение или имя переменной. Отладчик будет отображать его значение.

2. **Call Stack** — вложенные вызовы (иначе называют «стек вызовов»).

На текущий момент видно, отладчик находится в функции `pow` (`pow.js`, строка 6), вызванной из анонимного кода (`index.html`, строка 13).

3. **Scope Variables** — переменные.

На текущий момент строка 6 ещё не выполнялась, поэтому `result` равен `undefined`.

В `Local` показываются переменные функции: объявленные через `var` и параметры. Вы также можете там видеть переменную `this`, она создаётся автоматически. Если вы не знаете, что она означает — ничего страшного, мы это обсудим в главе [Контекст this в деталях \[23\]](#).

В `Global` — глобальные переменные, которые находятся вне функций.

Управление выполнением

Пришло время погонять скрипт и «оттрейсить» (от англ. trace, отслеживать) его работу.

Обратим внимание на панель управления в ней есть 4 основные кнопки:



— **продолжить выполнение (F8).**

Если скрипт не встретит новых точек остановки, то на этом работа в отладчике закончится.

Нажмите на эту кнопку.

Вы увидите, что отладчик остался на той же строке, но в Call Stack появился новый вызов. Это произошло потому, что в 6й строке находится рекурсивный вызов функции row, т.е. управление перешло в неё опять, но с другими аргументами.

Походите по стеку вверх-вниз — вы увидите, что действительно аргументы разные.



— **сделать шаг, не заходя внутрь функции (F10).**

Выполняет одну строку скрипта. Если в ней есть вызов функции — то отладчик обходит его стороной, т.е. не переходит на код внутри.

Нажмите на эту кнопку.

Отладчик перейдёт на строку 7. Всё правильно, хотя есть и тонкость.

Дело в том, что во вложенном вызове row есть брейкпойнт, а на включённых брейкпойнтах отладчик останавливается всегда. Даже если вложенный вызов и нажата эта кнопка.

...Но в данном случае вложенный вызов будет с `n=1`, поэтому сработает `if` и до строки 6 управление не дойдёт. Поэтому и остановки нет.



— **сделать шаг (F11).**

Переходит на следующую строку кода. Если есть вложенный вызов, то внутрь функции.



— **выполнять до выхода из текущей функции (Shift + F11).**

Как только текущая функция завершилась, отладчик тут же останавливает скрипт.

Удобно в случае, если мы нечаянно вошли во вложенный вызов, который нам совсем не интересен — чтобы быстро из него выйти.



— **отключить/включить все точки остановки.**

Эта кнопка никак не двигает нас по коду, она позволяет временно отключить все точки остановки в файле.

Процесс отладки заключается в том, что мы останавливаем скрипт, смотрим, что с переменными, переходим дальше и ищем, где поведение отклоняется от правильного.




Дополнительные возможности

Правый клик на номер строки позволяет запустить выполнение кода до неё (Continue to here).

Это очень удобно, если промежуточные строки нас не интересуют.

Консоль

При отладке, кроме просмотра переменных, бывает полезно запускать команды JavaScript. Для этого нужна консоль.

В неё можно перейти, нажав кнопку «Console» вверху-справа, а можно и открыть в дополнение к отладчику, нажав на кнопку  или клавишей ESC.

Самая любимая команда разработчиков: `console.log(...)`.

Она пишет переданные ей аргументы в консоль, например:

```
1 // результат будет виден в консоли
2 for(var i=0; i<5; i++) {
3   console.log("значение", i);
4 }
```

Полную информацию по специальным командам консоли вы можете получить на странице <http://firebug.ru/commandline.html>. Эти команды также действуют в Firebug (отладчик для браузера Firefox).

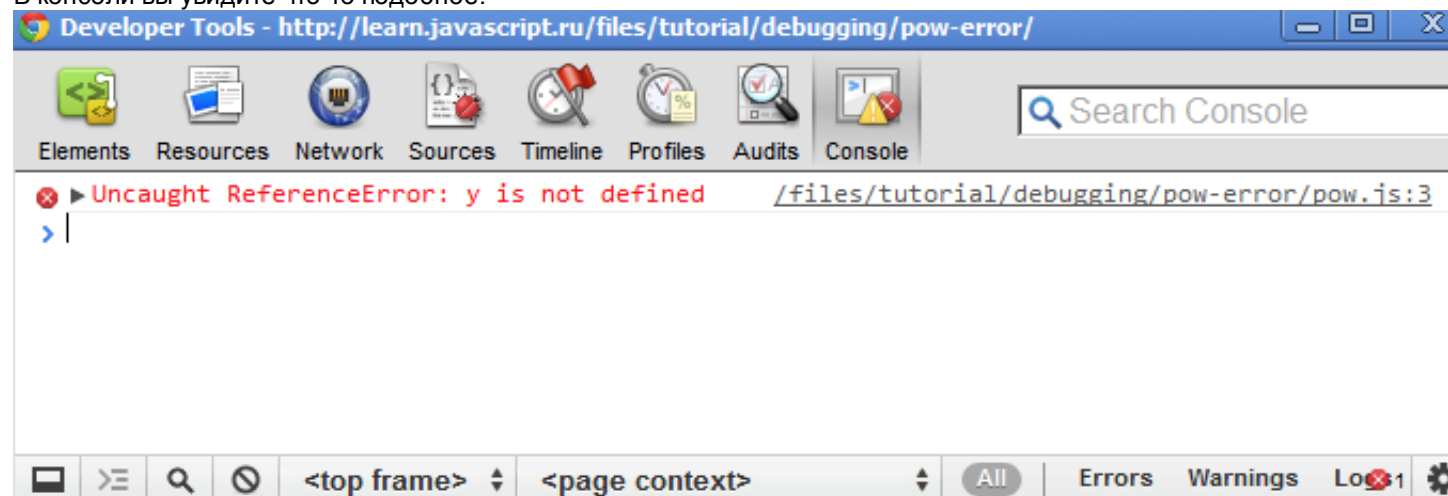
Консоль поддерживают все браузеры, но IE<10 поддерживает не все функции. Логирование работает везде, пользуйтесь им вместо alert.


Ошибки

Ошибки JavaScript выводятся в консоли.

Например, откройте страницу tutorial/debugging/pow-error/index.html [24]. Предыдущую отладку можно прекратить (очистить брейкпойнты  и затем продолжить ).

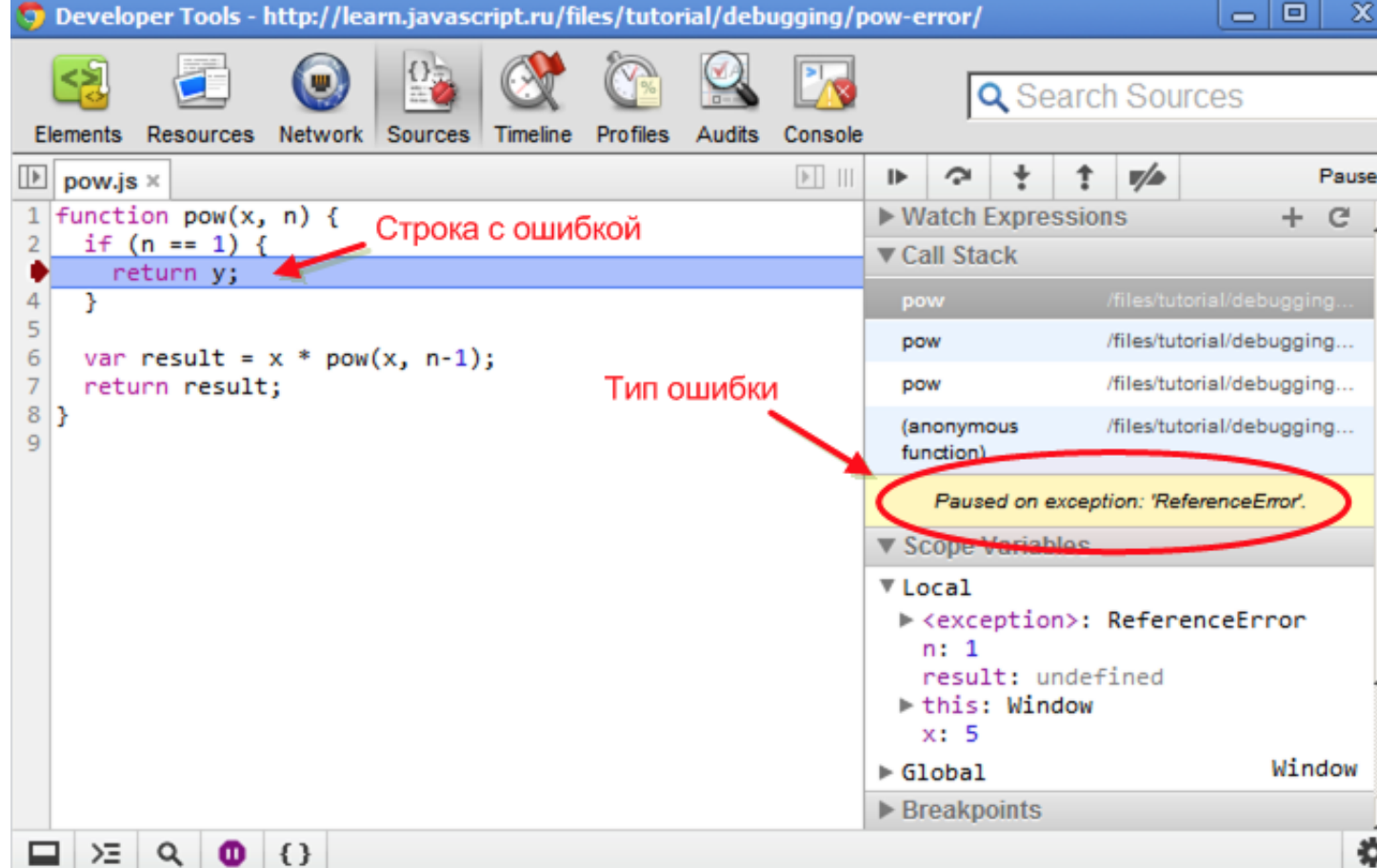
В консоли вы увидите что-то подобное:



Да, действительно, это потому что в этом скрипте есть ошибка. Но в чём же дело? Давайте посмотрим, какие были значения переменных на момент отладки. Для этого заставим отладчик остановиться при ошибке, нажав два раза на  (до фиолетового цвета).

Кстати, чтобы увидеть эту кнопку, нужно быть именно во вкладке Sources, не в Console.

Теперь перезагрузите страницу. Отладчик остановится на строке с ошибкой:



Можно посмотреть значения переменных. Поставить брейкпойнты раньше по коду и посмотреть, что привело к такой печальной картине. В данном случае-то всё просто: опечатка в имени переменной `y` вместо `x`. Этот тип ошибки называется `ReferenceError`.

Итого

Отладчик позволяет:

- ➔ Останавливаться на отмеченном месте (breakpoint) или по команде `debugger`.
- ➔ Выполнять код — по одной строке или до определённого места.
- ➔ Смотреть переменные, выполнять команды в консоли и т.п.

В этой статье кратко описаны возможности отладчика Google Chrome, относящиеся именно к работе с кодом.

Пока что это всё, что нам надо, но, конечно, инструменты разработчика умеют много чего ещё. В частности, вкладка `Elements` — позволяет работать со страницей (понадобится позже), `Timeline` — смотреть, что именно делает браузер и сколько это у него занимает и т.п.

Осваивать можно двумя путями:

1. [Официальная документация \[25\]](#) (на англ.)
2. Кликать правой кнопкой и двойным кликом в разных местах и смотреть, что получается.

Решения задач



Решение задачи: Ошибки в стиле

Вы могли заметить следующие недостатки:

1. Отсутствуют пробелы — между параметрами, вокруг операторов, при вложенном вызове `alert(pow(...))`.
2. Переменные `x` и `n` присвоены без `var`.
3. Фигурные скобки расположены на отдельной строке.
4. Логически разные фрагменты кода: ввод данных `prompt` и их обработка `if` не разделены вертикальным пробелом.
5. Строка с `alert` слишком длинная, лучше разбить её на две.
6. Не везде есть точки с запятой.



Решение задачи: Ошибки в стиле 2

Основная ошибка — неверный выбор имени функции `result`.

1. Во-первых, это существительное, а значит функцией быть не может (разве что какая-то особая договорённость о наименованиях).
2. Во-вторых, переменная `result` традиционно используется для хранения «текущего результата функции». Здесь эта традиция нарушена.

Как назвать функцию правильно? Один из вариантов — префикс `do`, т.е. `doPow`, он означает что эта функция как раз и делает реальную работу.

Также не лучший вариант — проверка с комментарием. Обычно код становится более читабельным, если выносить неочевидные действия в новую функцию. В этом случае имя этой функции послужит комментарием.

P.S. Рекурсия при возведении в степень — не лучший выбор, обычный цикл будет быстрее, но это скорее недочёт логики, а не ошибка стиля.

Исправленный код:

```
01 function pow(x, n) {  
02   if (!isNatural(n)) {  
03     return NaN;  
04   }  
05  
06   return doPow(x, n);  
07  
08   // ----  
09   function isNatural(n) {  
10     return n >= 1 && n == Math.round(n);  
11   }  
12  
13   function doPow(x, n) {  
14     return (n == 1) ? x : x*doPow(x, n-1);  
15   }  
16  
17 }
```

Ссылки

1. Старом стандарте <http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf>
2. Имена функций <http://learn.javascript.ru/function-basics#function-naming>
3. Имена переменных <http://learn.javascript.ru/variables#variable-naming>
4. Проверки в циклах лучше делать через «continue» <http://learn.javascript.ru/break-continue#continue>
5. JSDoc <http://en.wikipedia.org/wiki/JSDoc>
6. Aptana <http://aptana.com>
7. JetBrains <http://www.jetbrains.com/>
8. UML http://ru.wikipedia.org/wiki/Unified_Modeling_Language
9. Google JavaScript Style Guide <http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml>
10. JQuery Core Style Guidelines http://docs.jquery.com/JQuery_Core_Style_Guidelines

11. Idiomatic.JS <https://github.com/rwldrn/idiomatic.js>
12. Перевод https://github.com/rwldrn/idiomatic.js/tree/master/translations/ru_RU
13. Dojo Style Guide <http://dojotoolkit.org/community/styleGuide>
14. JSLint <http://www.jshint.com/>
15. Стилю JSLint <http://www.jshint.com/lint.html>
16. JSHint <http://www.jshint.com/>
17. Closure Linter <https://developers.google.com/closure/utilities/>
18. Google JavaScript Style Guide <http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml>
19. How To Write Unmaintainable Code <http://mindprod.com/jgloss/unmain.html>
20. Wrap <http://api.jquery.com/wrap/>
21. Дао дэ цзин <http://lib.ru/POECHIN/lao1.txt>
22. Tutorial/debugging/pow/index.html <http://learn.javascript.ru/files/tutorial/debugging/pow/index.html>
23. Контекст this в деталях <http://learn.javascript.ru/this>
24. Tutorial/debugging/pow-error/index.html <http://learn.javascript.ru/files/tutorial/debugging/pow-error/index.html>
25. Официальная документация <https://developers.google.com/chrome-developer-tools/docs/overview>