



# Обработка ошибок

# Содержание

<b>1. Виды ошибок</b>	<b>3-4</b>
◦ Логические ошибки	4
◦ Ошибки выполнения	4
<b>2. Контекст выполнения</b>	<b>5-9</b>
◦ Область видимости	6
◦ this	6
<b>3. Объект Error</b>	<b>10-12</b>
◦ Параметры	11
◦ Стандартные свойства	11
<b>4. throw, try, catch</b>	<b>13-15</b>
◦ Пример: обработка определённой ошибки	14
<b>5. Консоль в Chrome Developer Tools</b>	<b>16-21</b>
◦ Маски вывода	18
◦ console.trace()	18
◦ console.assert()	19
◦ console.group() и console.groupEnd();	19
◦ console.time() и console.timeEnd()	20
◦ Фильтр сообщений	20
◦ \$(), \$\$() и \$x()	21
◦ \$0 — \$4	21
◦ \$_	21
<b>5. Точки остановки (Breakpoints)</b>	<b>22-28</b>
◦ Условные точки остановки (Conditional breakpoints)	24
◦ Точки остановки DOM (DOM Breakpoints)	25
◦ Точки остановки XHR (XHR Breakpoints)	26
◦ Точки остановки по событиям (Event Listener Breakpoints)	27
◦ Оператор debugger;	28

# 1

## Виды ошибок

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

```
function func({  
  
}
```

## Логические ошибки

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

```
var number = '2';  
  
for (var i = 1; i < 5; i++) {  
    console.log(i + number); //12  
                                //22  
                                //32  
}
```

## Ошибки выполнения

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

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

# 2

## **Контекст выполнения**

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

## Область видимости

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

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

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

Все изменения локальных переменных являются изменениями свойств этого неявного объекта.

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

## this

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

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

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

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

*Код в JavaScript может быть одного из следующих типов:*

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

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

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

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

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

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

### **Главные моменты:**

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



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

# 3

## Объект Error

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

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

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

## Параметры

**message** - Необязательный параметр. Человеко-читаемое описание ошибки.

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

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

## Стандартные свойства

**message** - описание ошибки.

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

```
var e = new Error("Произошла проблема");  
  
console.log(e.message); //произошла проблема
```

**name** - название типа ошибки.

По умолчанию, объекты класса Error получают значение "**Error**". Однако, можно его поменять на другое:

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

console.log(e.name); //выведет Error

e.name = "ParseError"
```

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

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

# 4

**throw, try,  
catch**

**Ключевое слово *throw*** используется для выбрасывания исключения.

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

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

## Пример: обработка определённой ошибки

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

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

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

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

# 5

## **Консоль в Chrome Developer Tools**



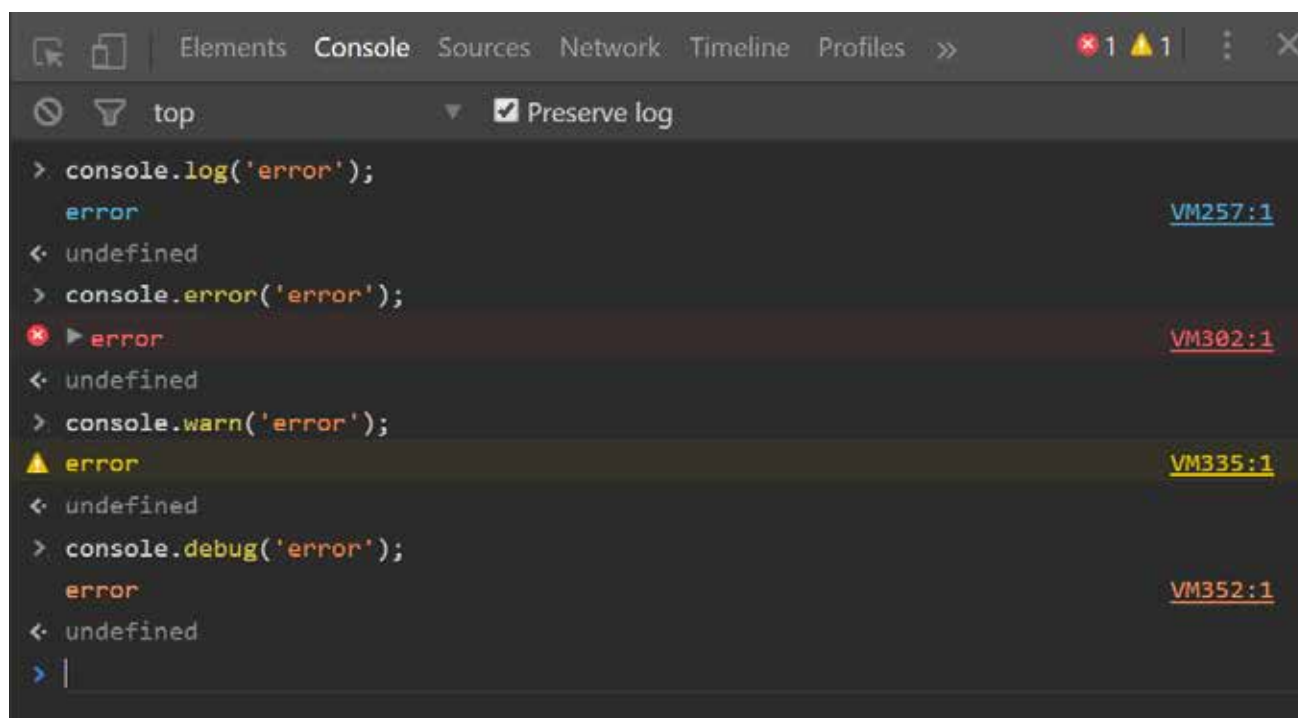
**Консоль в Chrome Developer Tools** - интерактивный JavaScript-интерпретатор. Консоль дает разработчику доступ к ряду удобных и полезных функций для логирования и отображения ошибок

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

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

Отличаются классификацией выводимых сообщений:

<code>error()</code>	помечает сообщения как ошибки
<code>warn()</code>	помечает сообщения как предупреждения
<code>debug()</code>	помечает сообщения как отладочные



The screenshot shows the Chrome Developer Tools Console with the 'Console' tab selected. The 'top' filter is active, and the 'Preserve log' checkbox is checked. The console displays four log entries:

- A log entry for `console.log('error');` showing the text 'error' in blue, with a link to `VM257:1`.
- An error entry for `console.error('error');` showing the text 'error' in red, a red 'x' icon, and a link to `VM302:1`.
- A warning entry for `console.warn('error');` showing the text 'error' in yellow, a yellow triangle icon, and a link to `VM335:1`.
- A debug entry for `console.debug('error');` showing the text 'error' in orange, and a link to `VM352:1`.

Each log entry is followed by the text `< undefined` on the next line.

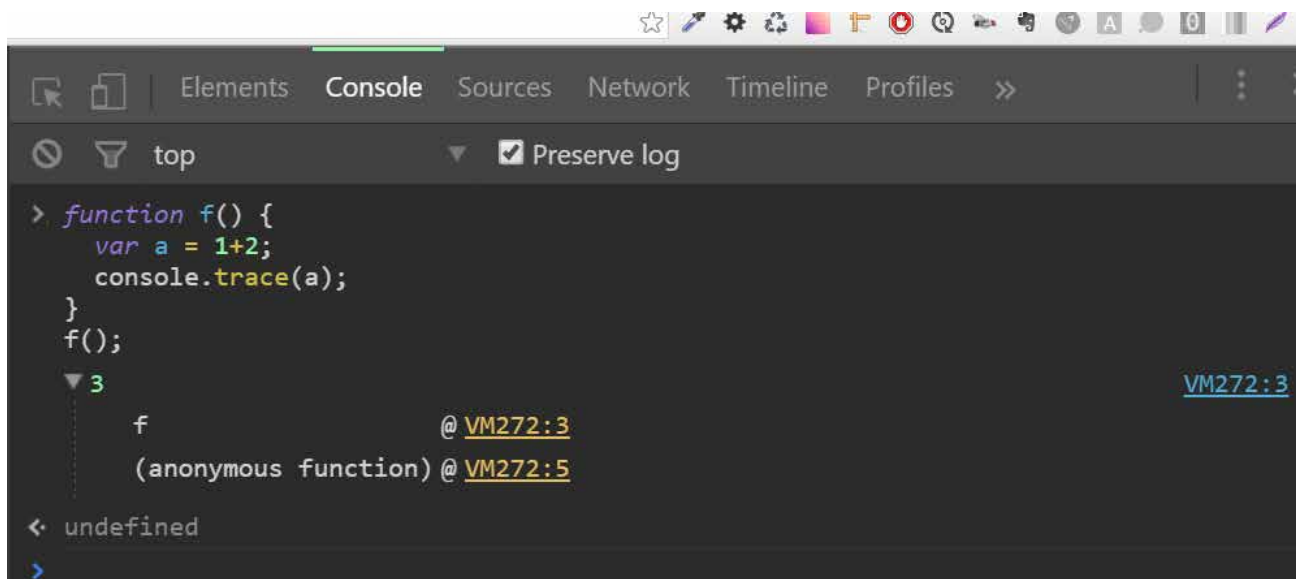
## Маски вывода

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

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

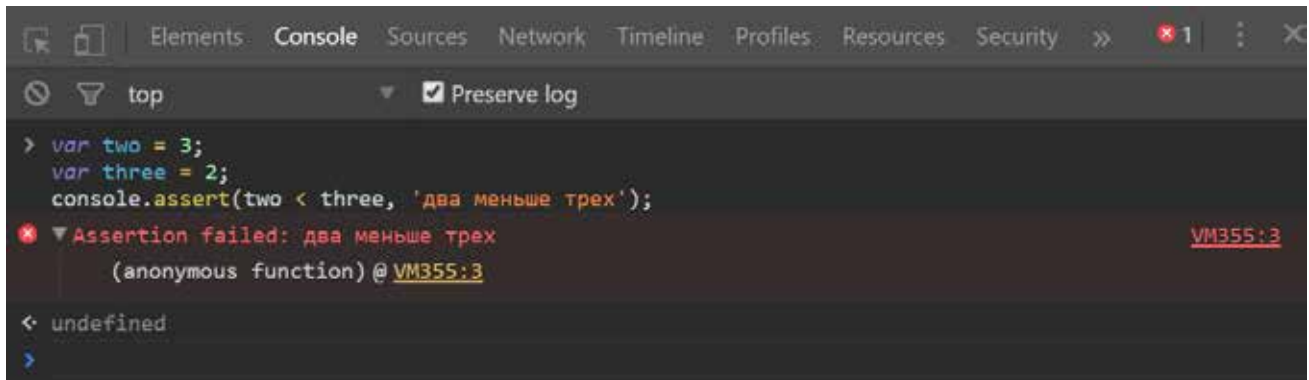
## console.trace()

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



## console.assert()

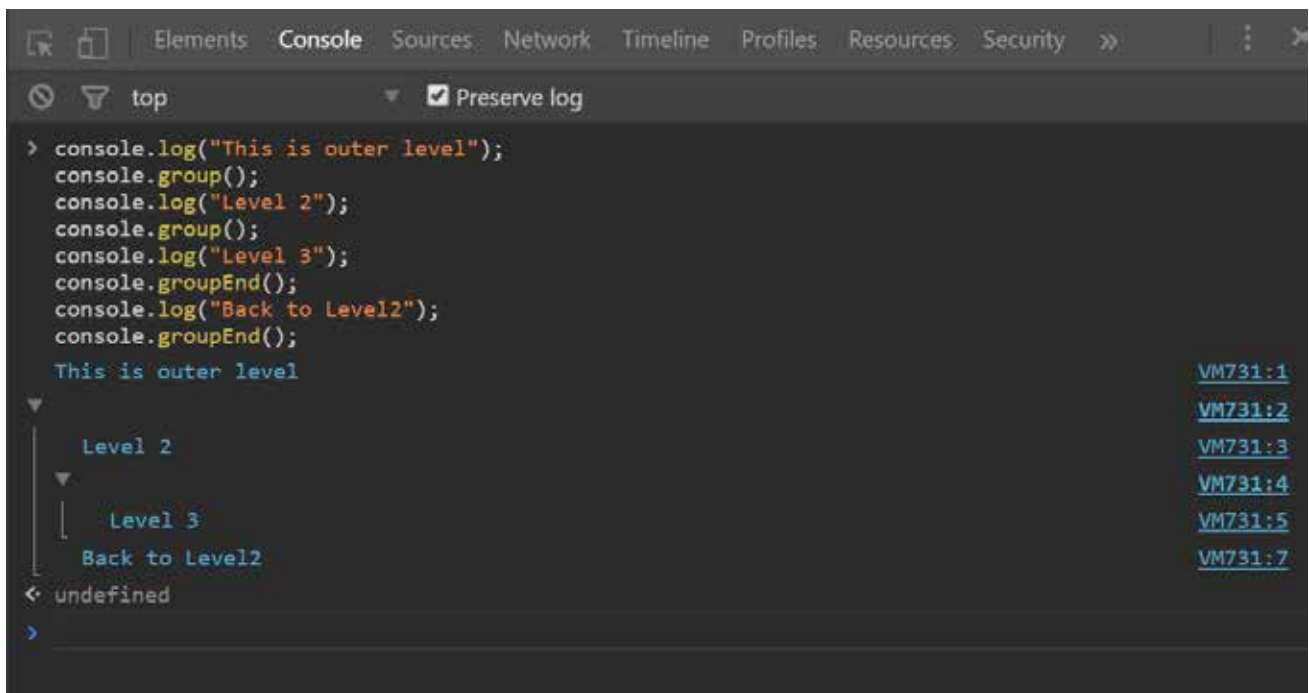
Функция `assert` проверяет выражение, переданное первым параметром, и если выражение ложно, записывает в консоль ошибку вместе со стеком вызовов:



```
> var two = 3;
    var three = 2;
    console.assert(two < three, 'два меньше трех');
✖ Assertion failed: два меньше трех VM355:3
  (anonymous function) @ VM355:3
< undefined
>
```

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

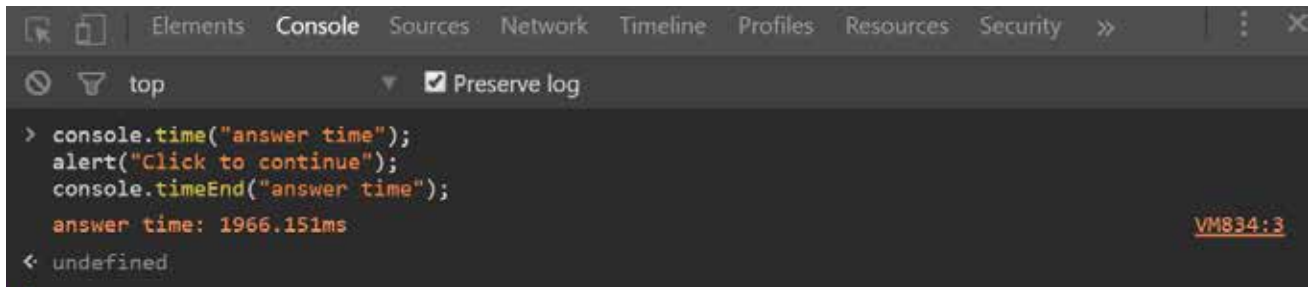
Функции для группировки вывода. Функция **group()** открывает группу сообщений, в качестве параметра принимает название группы (поддерживается форматирование, как в **console.log()**). **groupEnd()** закрывает группу



```
> console.log("This is outer level");
    console.group();
    console.log("Level 2");
    console.group();
    console.log("Level 3");
    console.groupEnd();
    console.log("Back to Level2");
    console.groupEnd();
This is outer level
└─ Level 2
   └─ Level 3
      Back to Level2
< undefined
>
```

## console.time() и console.timeEnd()

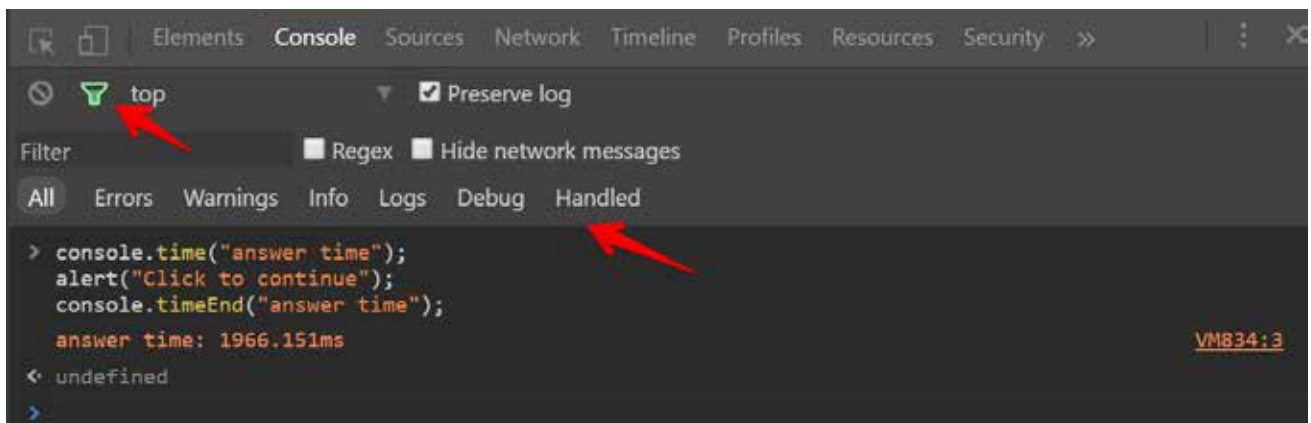
Функции для вычисления времени исполнения кода. Функция time() запускает таймер, а функция timeEnd() останавливает таймер и выводит его значение. Обе функции принимают название таймера в качестве обязательного параметра.



```
> console.time("answer time");
alert("Click to continue");
console.timeEnd("answer time");
answer time: 1966.151ms
< undefined
```

## Фильтр сообщений

На вкладке консоли расположен фильтр сообщений по типу.



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

## \$(), \$\$() и \$x()

Функции, упрощающие выборку элементов, работают только в консоли.

**Функция `$()`** возвращает первый элемент, соответствующий переданному селектору. Вторым параметром можно передать контекст поиска:

Функция **`$$()`** аналогична **`$()`**, но возвращает все найденные элементы

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

## \$0 — \$4

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

## \$\_

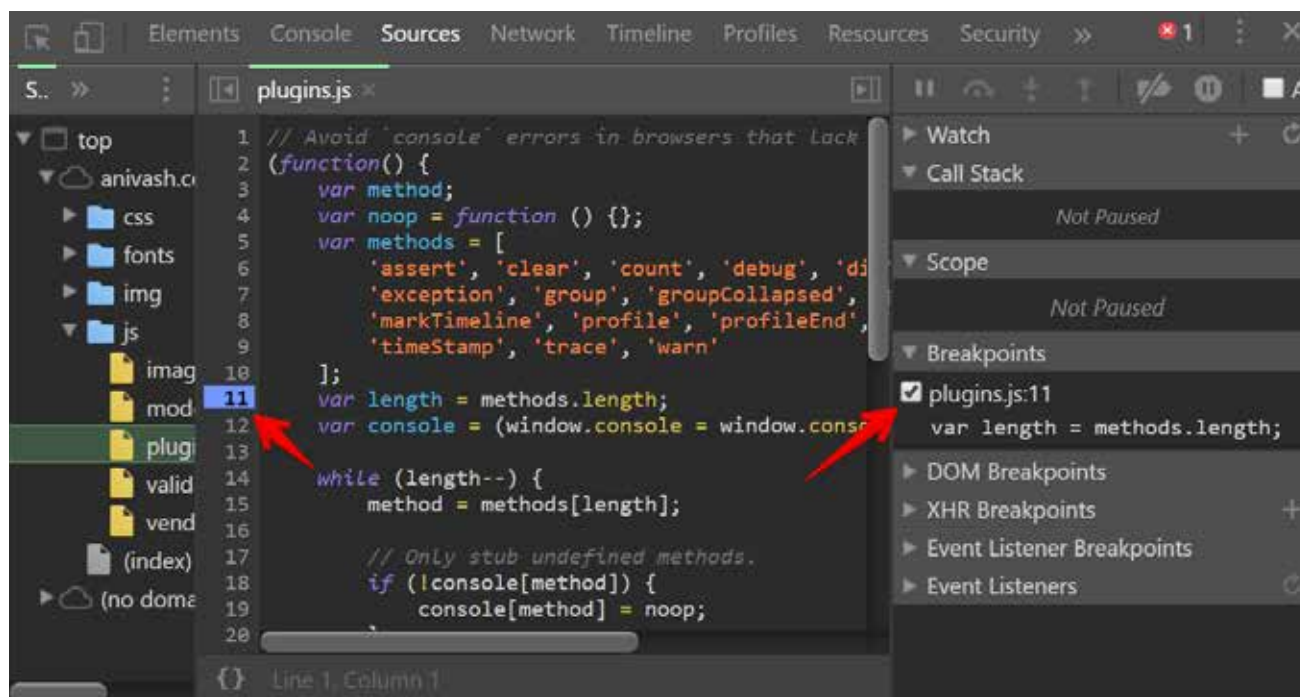
Переменная **`$_`** хранит результат отработки последней команды в консоли. Это позволяет использовать результат выполнения одной команды в другой команде. Попробуйте выполнить эти команды по очереди:

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

# 6

## **Точки остановки (Breakpoints)**

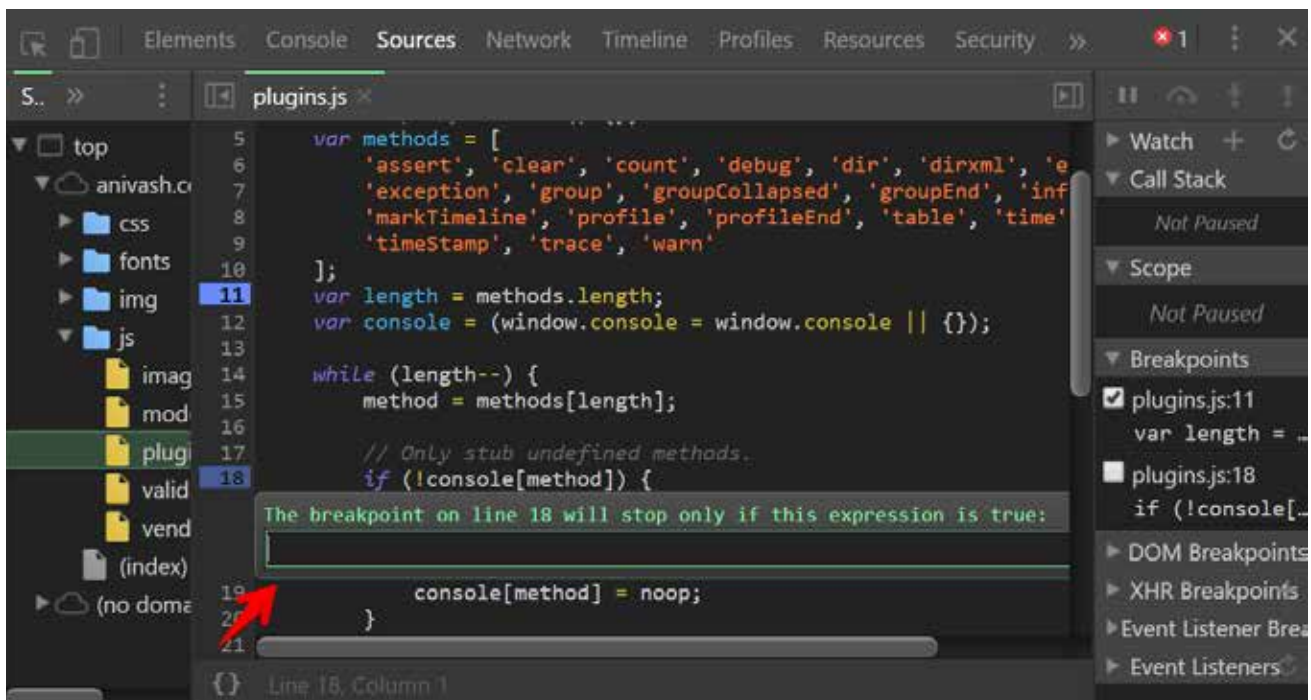
Самый простой вариант остановки кода — задать обычную точку остановки. Точка остановки привязывается к строке. Как только интерпретатор JavaScript достигает этой строки, код встает на паузу. В режиме паузы можно посмотреть значения всех переменных, как локальных, так и глобальных, а также исполнять код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



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

## Условные точки останова (Conditional breakpoints)

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

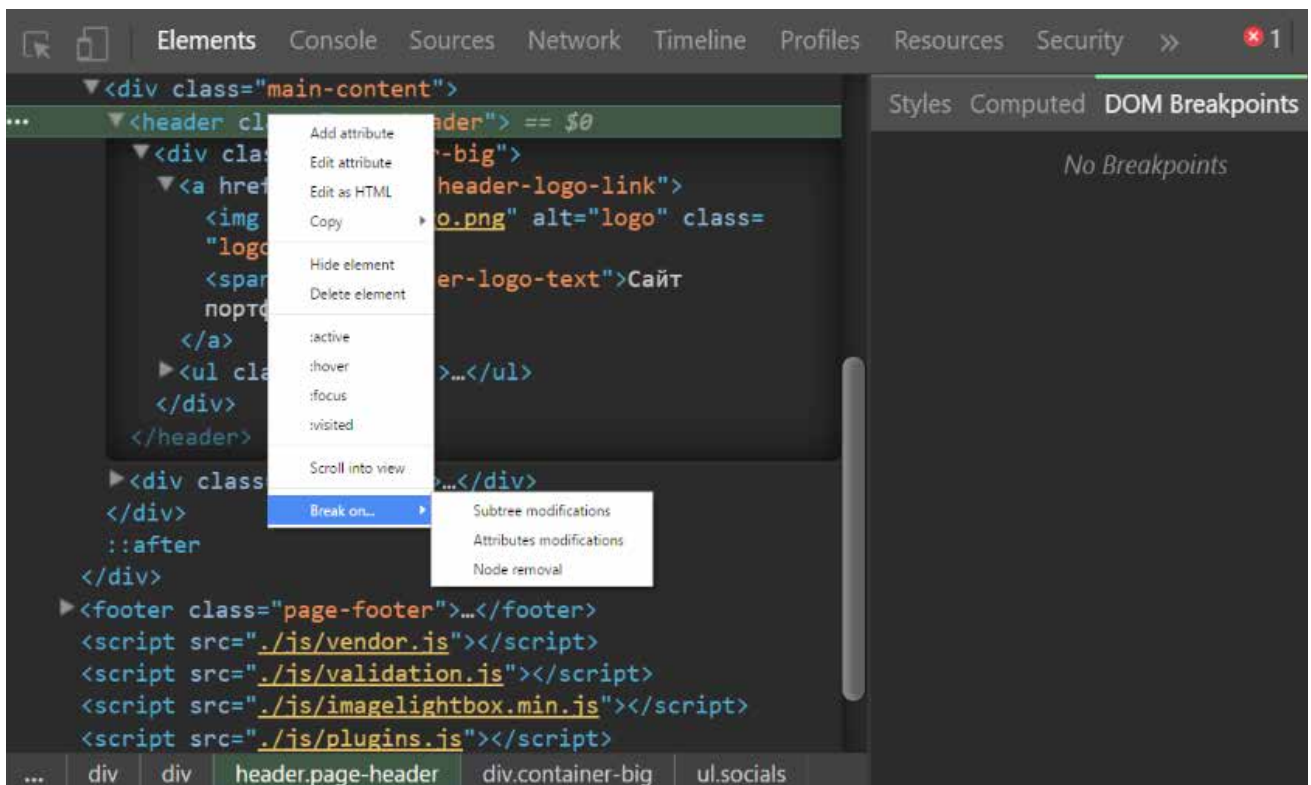


Чтобы создать условную точку останова, нужно кликнуть правой кнопкой мышки по номеру строки, выбрать «Add conditional breakpoint...» и ввести выражение. Если в момент исполнения кода выражение истинно, исполнение будет приостановлено.



## Точки остановки DOM (DOM Breakpoints)

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

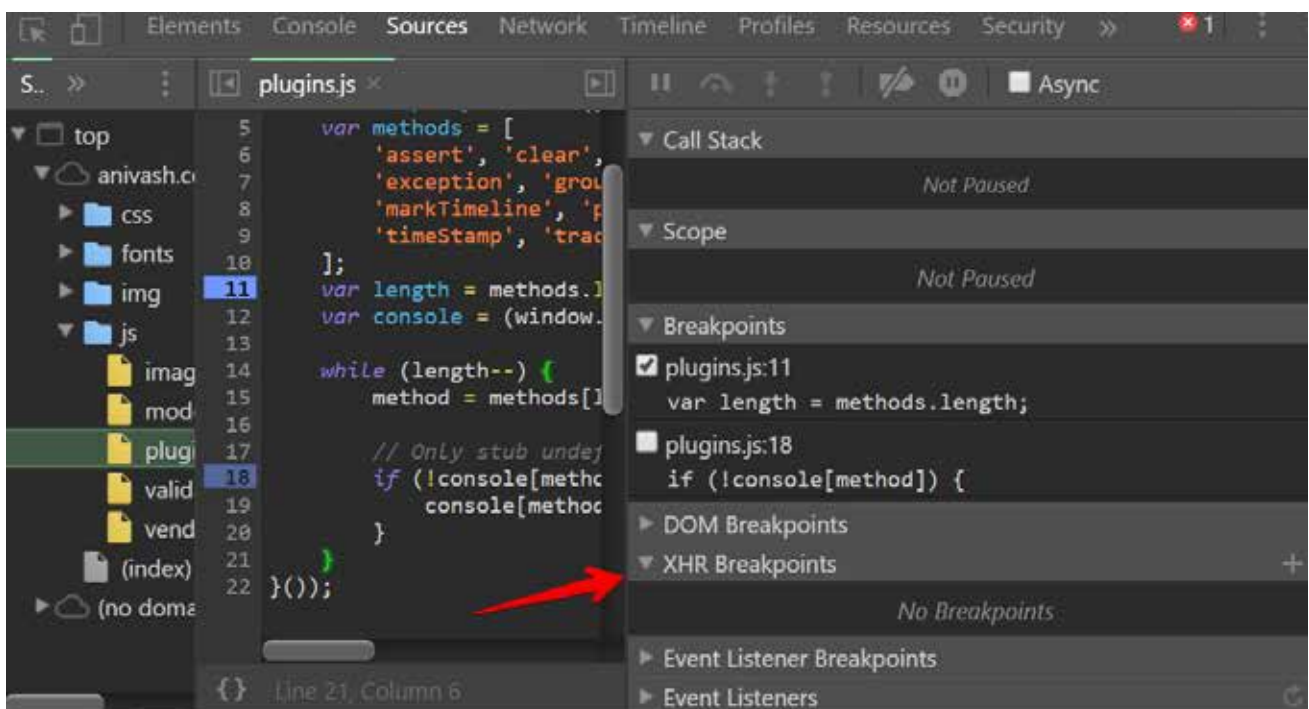


Для создания точки остановки DOM необходимо найти элемент во вкладке **«Elements»**, щелкнуть его правой кнопкой мыши и в открывшемся меню выбрать пункт **«Break on...»**.

## Точки остановки XHR (XHR Breakpoints)

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

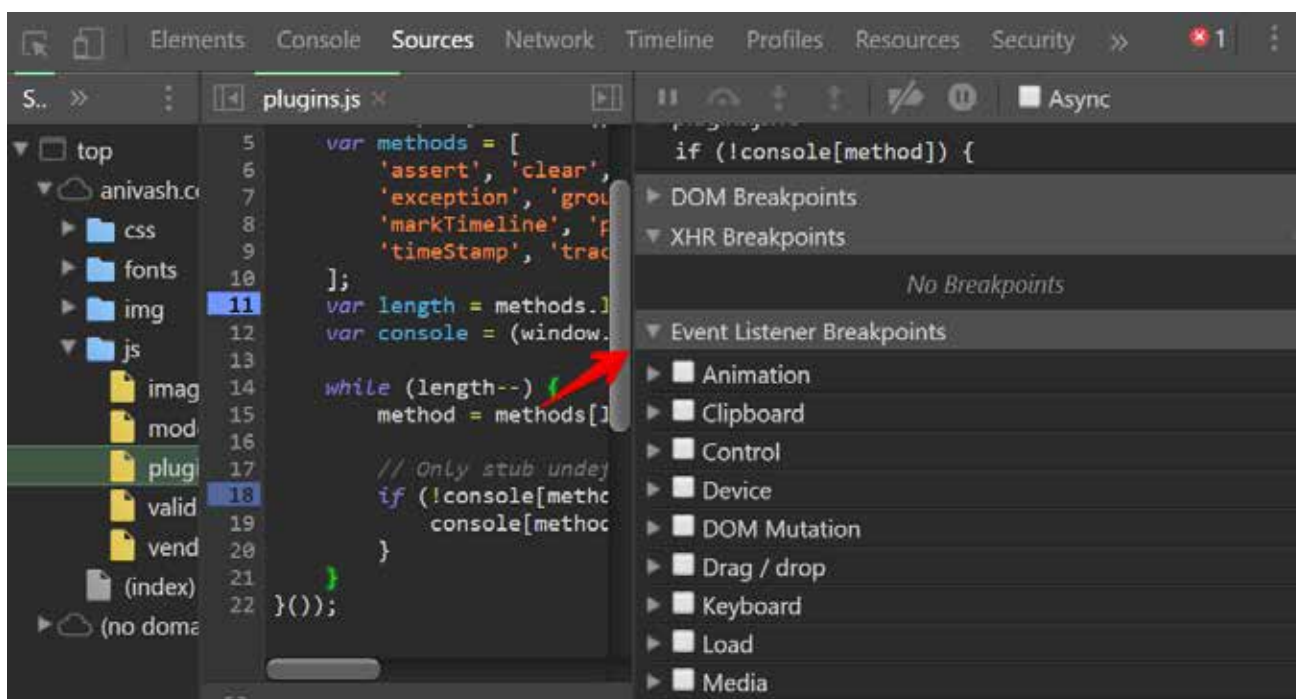
Установить точку остановки XHR можно со вкладки «Sources», нажав на иконку плюсика напротив заголовка «XHR Breakpoints» в правой панели.



## Точки остановки по событиям (Event Listener Breakpoints)

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

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



## Оператор debugger;

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

