

# Module 6: Troubleshooting JavaScript Code

# Agenda

---

1. Exception Handling
2. Debugging Code in Browser
3. Using Console API
4. Useful links

---

# Exception Handling

# Errors are Natural

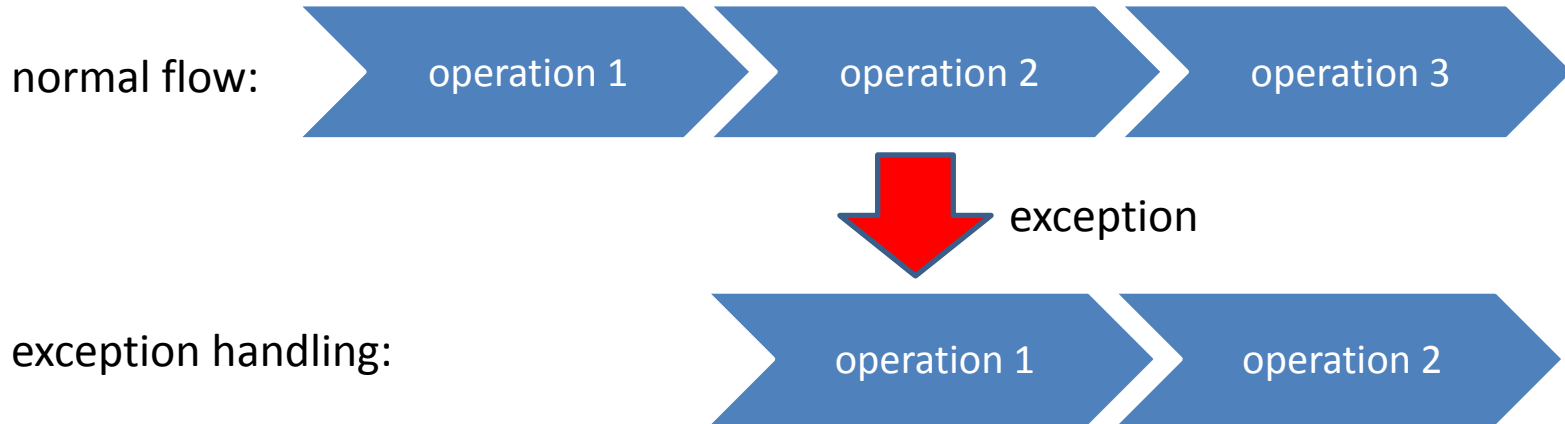
- Any software solution faces **errors**: invalid user input, broken connection or bugs in code
- Errors break normal flow of the program execution and may lead to fatal results in case if not handled properly



[www.fasticon.com](http://www.fasticon.com)

# What is Exception and Exception Handling?

- **Exception** – is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- **Exception handling** is convenient way to handle errors



# Exception Syntax

To make exception handling possible we should use two keywords: **try** and **catch**:

```
try {  
    // Block of code that may  
    // raise an exception  
} catch (err) {  
    // Block of code to  
    // handle an exception  
}
```

# Throwing Exceptions

To raise an exception we use **throw** keyword.

Throwing an exception will break normal code execution on a line when the keyword is met and will give control to the nearest catch block.

Syntax:

```
throw (new Error("Some meaningful message"));
```

# Exception Handling Sample

- In a sample below we ask the user to enter age and convert it to a number. If conversion returns *NaN* we throw an exception and handle it with catch block.
- Note that JS itself does not throw exceptions on math errors, it returns NaN

```
01 var age = parseInt(window.prompt("Enter your age"));
02 try {
03     if (isNaN(age))
04         throw (new Error("You entered incorrect value!"));
05     var nextAge = age + 10;
06     alert("In ten years you will be " + nextAge);
07 }
08 catch (err) {
09     alert(err.message);
10 }
```



# Using finally keyword

Keyword **finally** is used in **try..catch** construction to define block of code that is called whenever exception occurs or not.

The main purpose of it is to free resources allocated just before *try* keyword

```
try {  
    // Block of code that may raise an exception  
} catch (err) {  
    // Block of code to handle an exception  
} finally {  
    // Block of code that called whenever  
    // exception occurs or not  
}
```

# Method `.onerror()`

- Method **`window.onerror()`** called each time when unhandled exception occurs.
- The **`.onerror()`** event handler provides three pieces of information to identify the exact nature of the error:
  - **Error message.** The same message that the browser would display for the given error
  - **URL.** The file in which the error occurred
  - **Line number.** The line number in the given URL that caused the error

# Sample .onerror() handler

In a sample below we assign `.onerror()` event handler that is called on button click because there is no function as `myFunc()` defined on the page:

```
<html>
<head>
  <script type="text/javascript">
    window.onerror = function (msg, url, line) {
      alert("Message : " + msg);
      alert("url : " + url);
      alert("Line number : " + line);
    }
  </script>
</head>
<body>
  <p>Click the following to see the result:</p>
  <form>
    <input type="button" value="Click Me" onclick="myFunc();" />
  </form>
</body>
</html>
```

---

# Debugging Code in Browser

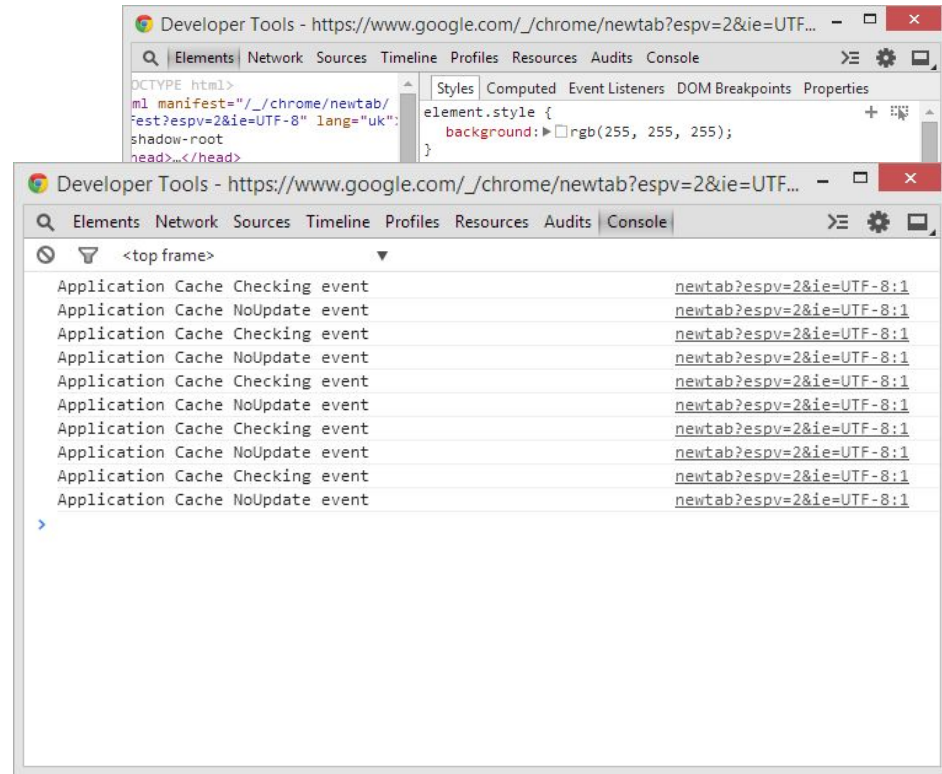
# What is Debugging?

- **Debugging** is a process of searching and removing *bugs* from the code
- The process of debugging might be not easy and sometimes becomes very tricky
- Writing clean, well-documented code that conforms coding conventions greatly simplifies debugging process
- Most modern browsers have built-in tools or addons for debugging JavaScript code



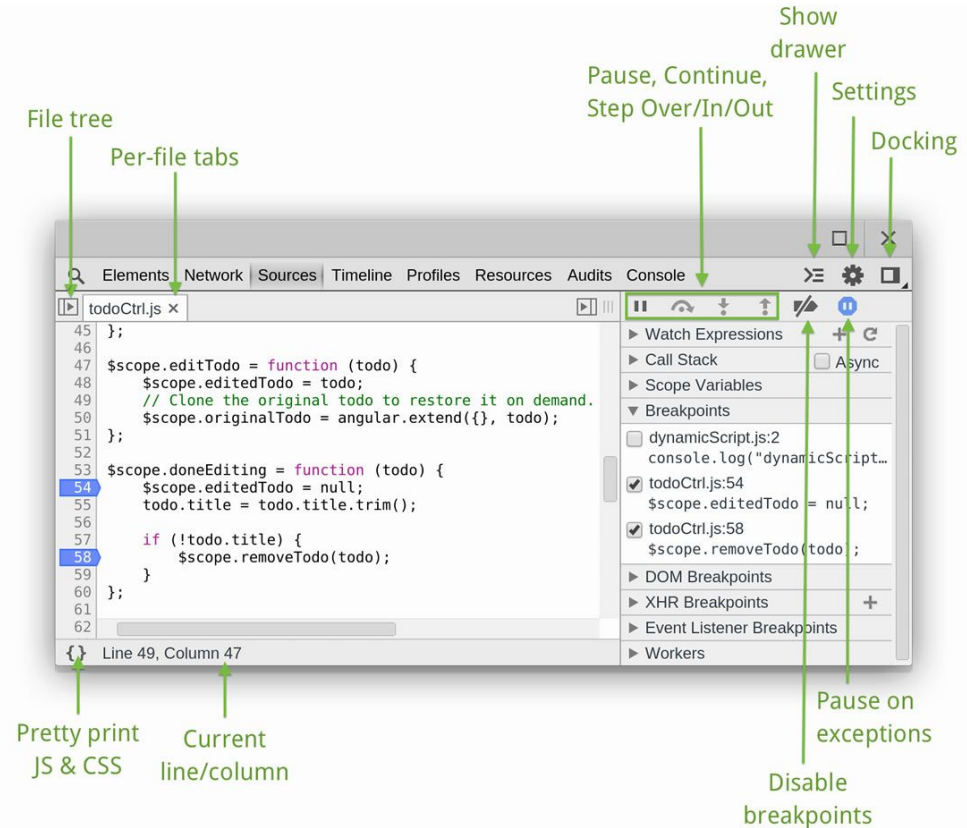
# Using Developers Tools

- Press **F12** to access **Developers Tools** in most browsers
- **Console tab** allows to execute JS code and see output of commands including error messages



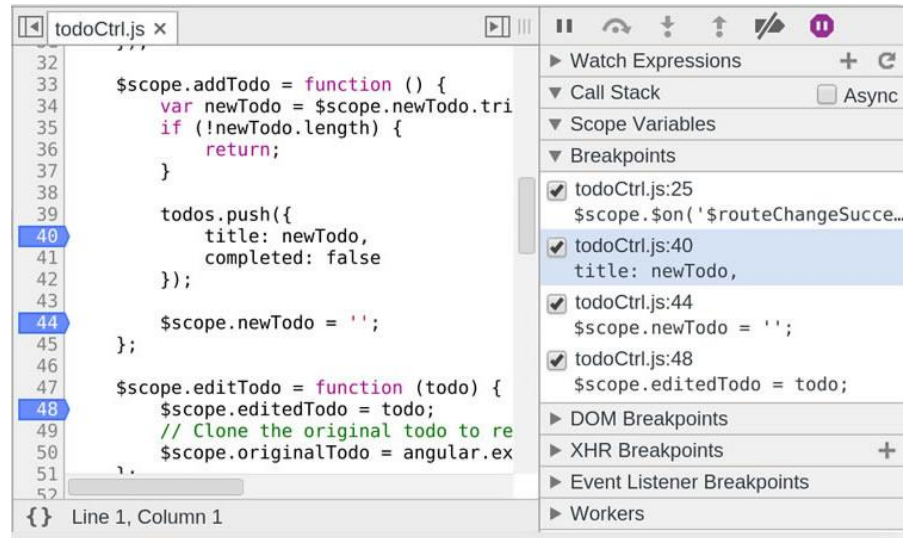
# Code Executions Controls in Chrome Browser

- Google Chrome browser provides full-featured debugger that has execution control buttons
- Opening Sources tab allows to choose JS code of a solution in external files as well as in inside html file



# Setting Breakpoints for JS Code in Chrome

- In Developer Tools open Sources tab and choose external JS file or navigate to JS code embedded In HTML file.
- Click the **line gutter** to set a breakpoint for that line of code, select another script and set another breakpoint.





# Execution Control Buttons in Chrome

- ▶ **Continue:** continues code execution to another breakpoint.
- ↺ **Step over:** step through code line-by-line, do not enters functions
- ⤵ **Step into:** acts like Step over, however clicking Step into at the function call will cause the debugger to move its execution to the first line in the functions definition.
- ⤴ **Step out:** allows to run current function till the end move debugger's execution to the parent function.
- 🚩 **Toggle breakpoints:** toggles breakpoints on/off while leaving their enabled states intact.

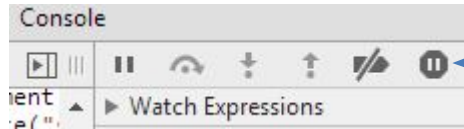
# Pause on Exceptions

There are two buttons to pause on exceptions:

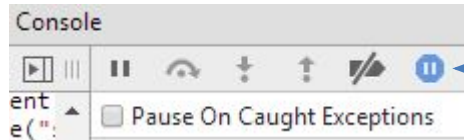
 – pause on all exceptions

 – pause on uncaught exceptions only

Second button becomes visible only if first is pressed



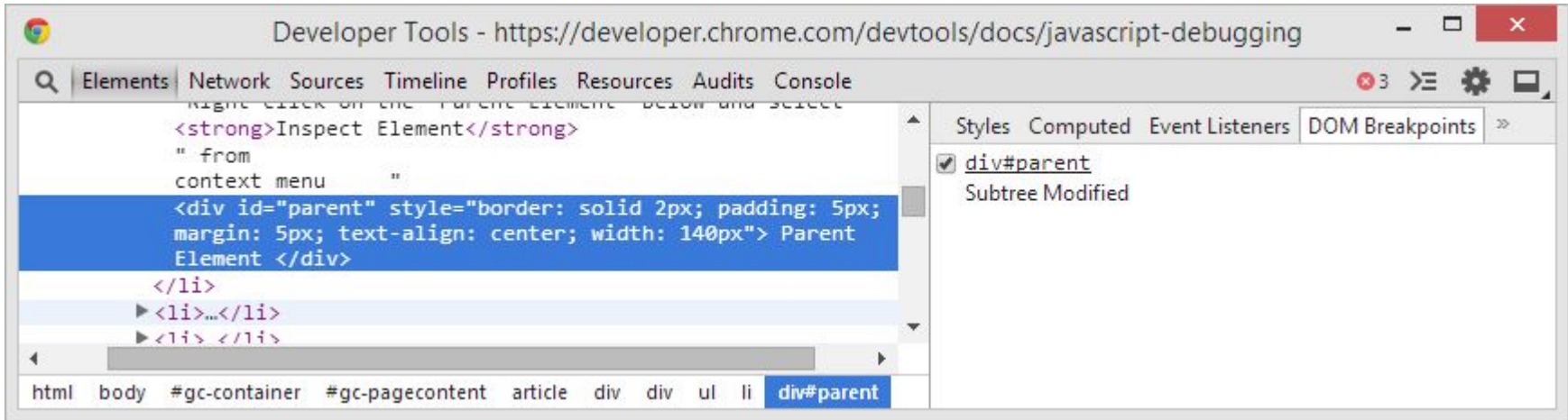
1. Pause on all exceptions



2. Pause on uncaught exceptions only

# Breakpoints on DOM Mutation Events

To stop execution on DOM mutation events right click on element, select Inspect Element, right click on highlighted element and select **Break on Subtree Modifications**

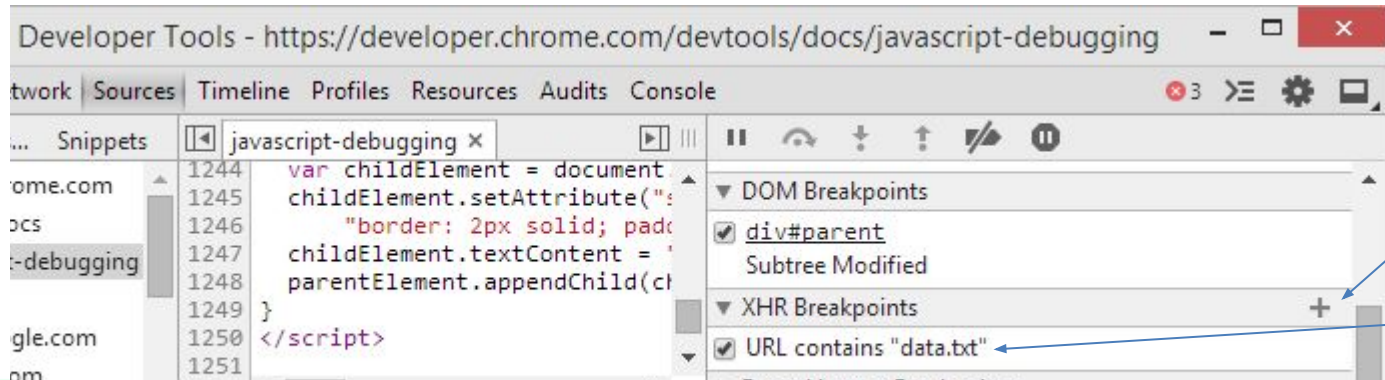


# Breakpoints on XMLHttpRequest Events

**XMLHttpRequest** object is used to make *Ajax* requests. We'll learn Ajax in detail in module 10.

To make breakpoints on XMLHttpRequest:

1. Click "+" button in XHR Breakpoints section;
2. Set URL filter in input field



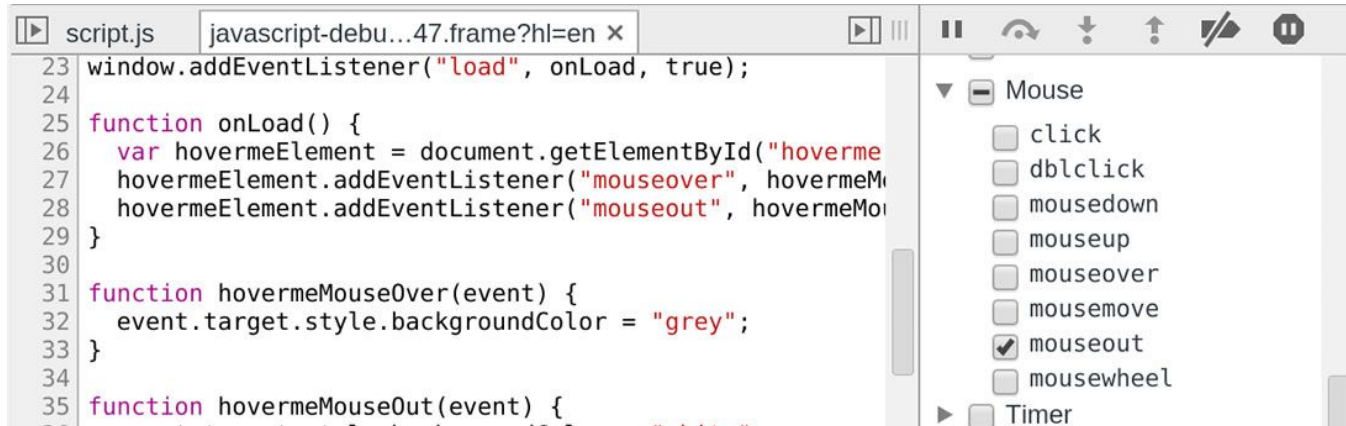
1. Click button

2. Set URL filter

# Breakpoints on JavaScript Event Listeners

To set breakpoint on Event Listeners:

- Expand **Event Listener Breakpoints** sidebar pane on the right side of **Scripts** panel
- Expand **Mouse** entry
- Set a mouseout Event Listener breakpoint by clicking on the checkbox near the **mouseout** entry



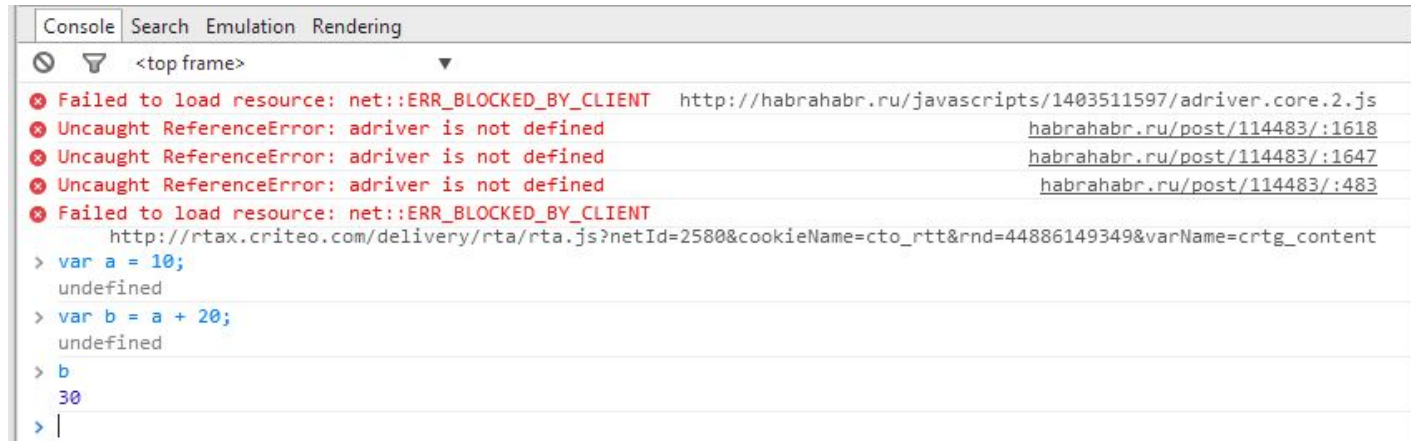
---

# Using Console API

# Console object

The **console object** provides access to the browser's debugging console. Console allows to log useful events and data while developing and debugging the solution.

Sample output of browser's console:



The screenshot shows a browser's developer console with the following content:

```
Console Search Emulation Rendering
<top frame>
Failed to load resource: net::ERR_BLOCKED_BY_CLIENT http://habrahabr.ru/javascripts/1403511597/adriver.core.2.js
Uncaught ReferenceError: adriver is not defined habrahabr.ru/post/114483/:1618
Uncaught ReferenceError: adriver is not defined habrahabr.ru/post/114483/:1647
Uncaught ReferenceError: adriver is not defined habrahabr.ru/post/114483/:483
Failed to load resource: net::ERR_BLOCKED_BY_CLIENT
http://rtax.criteo.com/delivery/rta/rta.js?netId=2580&cookieName=cto_rtt&rnd=44886149349&varName=crtg_content
> var a = 10;
undefined
> var b = a + 20;
undefined
> b
30
> |
```

# Useful Methods

Useful methods of **console** object:

- **.log()** – general output of logging information
- **.info()**, **.warn()**, **.error()** – same as log() but add notification icons
- **.dir()** – shows specific JS object
- **.dirxml()** – shows xml code or html code of DOM element
- **.group()**, **.groupCollapsed()**, **.groupEnd()** – grouping output
- **.time()**, **.timeEnd()** – setting timer
- **.profile()**, **.profileEnd()** – using profiling tools
- **.assert()** – asserting conditions



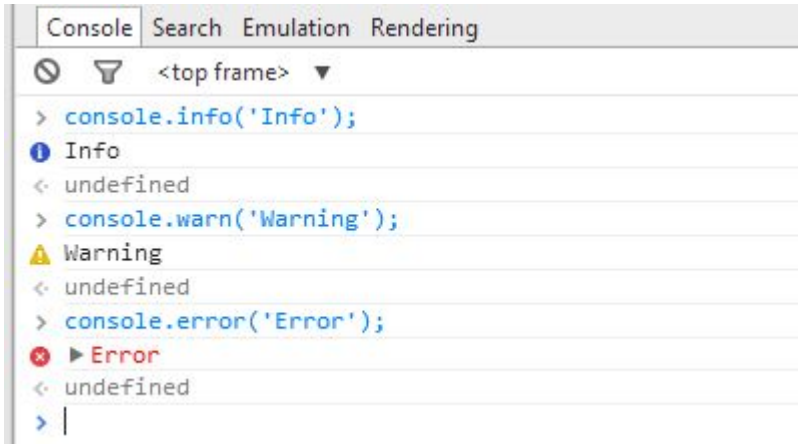
# Method .log()

- Method **.log()** used for general output of logging information
- Method accepts any number of arguments
- It is possible to use string formatting commands (%s – string, %d – decimal, %i – integer, %f – floating point)
- Sample:

```
console.log('Hello, my name is %s, my age is %i', 'John', 20);  
> Hello, my name is John, my age is 20
```

# Methods `.info()`, `.warn()`, `.error()`

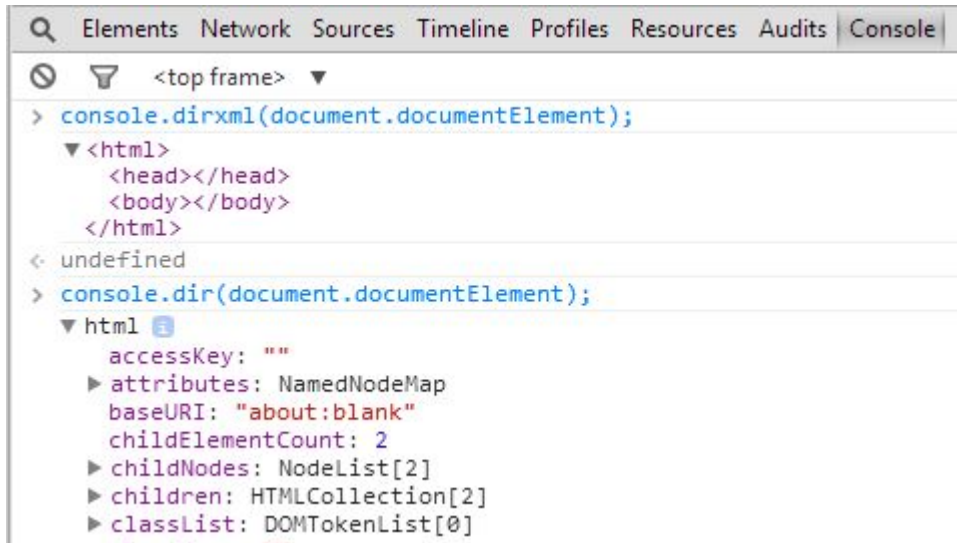
Methods `.info()`, `.warn()`, `.error()` act almost as `.log()` but add icons to console output that allows to distinguish between different type of output



```
Console Search Emulation Rendering
<top frame>
> console.info('Info');
Info
< undefined
> console.warn('Warning');
Warning
< undefined
> console.error('Error');
Error
< undefined
> |
```

# Methods .dirxml() and .dir()

Method **.dirxml()** – shows xml code or html code of DOM element, **.dir()** – shows specific JS object :

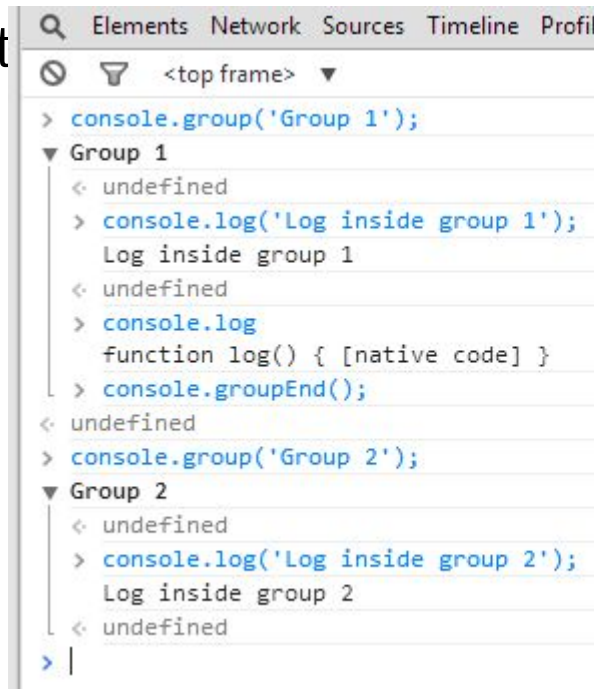


```
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
> console.dirxml(document.documentElement);
<html>
  <head></head>
  <body></body>
</html>
< undefined
> console.dir(document.documentElement);
html
  accessKey: ""
  ▶ attributes: NamedNodeMap
  baseURI: "about:blank"
  childElementCount: 2
  ▶ childNodes: NodeList[2]
  ▶ children: HTMLCollection[2]
  ▶ classList: DOMTokenList[0]
```

# Grouping Console Output

There are methods to group console output

- `.group()` – start group
- `.groupEnd()` – end group
- `.groupCollapsed()` – start group collapsed



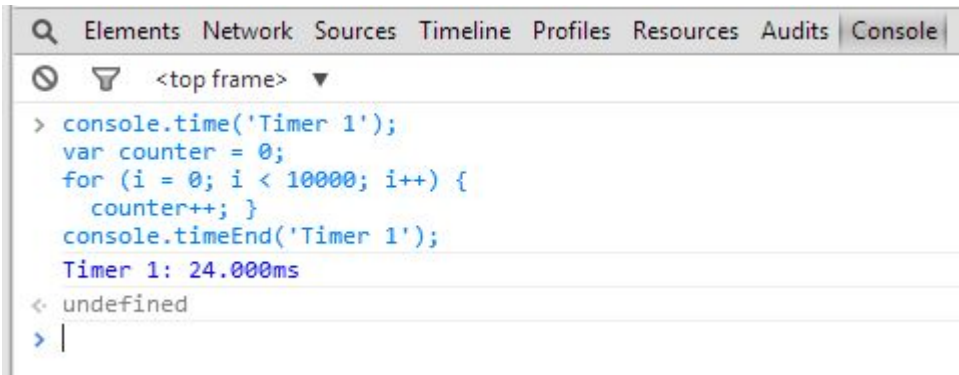
The screenshot shows a browser's developer console with the following code and output:

```
> console.group('Group 1');
Group 1
  < undefined
  > console.log('Log inside group 1');
  Log inside group 1
  < undefined
  > console.log
  function log() { [native code] }
  > console.groupEnd();
< undefined
> console.group('Group 2');
Group 2
  < undefined
  > console.log('Log inside group 2');
  Log inside group 2
  < undefined
> |
```

# Setting Timer

To measure execution time of code blocks use methods:

- `.time('Timer mark')` – start timer
- `.timeEnd('Timer mark')` – stop timer

A screenshot of a web browser's developer console. The console is open to the 'Console' tab, showing the execution of a JavaScript code block. The code starts with `console.time('Timer 1');`, followed by a loop that increments a counter from 0 to 10000. It then ends with `console.timeEnd('Timer 1');`. The console output shows the result: `Timer 1: 24.000ms`. Below the code, there is a prompt `< undefined` and a cursor on a new line.

```
Q Elements Network Sources Timeline Profiles Resources Audits Console
< top frame >
> console.time('Timer 1');
  var counter = 0;
  for (i = 0; i < 10000; i++) {
    counter++; }
  console.timeEnd('Timer 1');
  Timer 1: 24.000ms
< undefined
> |
```

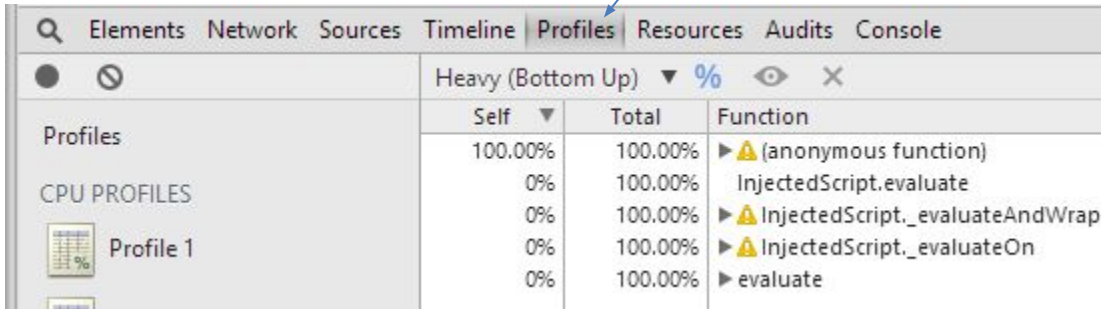
# Profiling Code

To display profiling stack use methods:

- `.profile()` – start profiler
- `.profileEnd()` – stop profiler

access  
profiling  
results here

```
Elements Network Sources Timeline Profiles R
<top frame>
> console.profile();
var counter = 0;
for (i = 0; i < 10000; i++) {
  counter++;
}
console.profileEnd();
Profile 'Profile 4' started.
Profile 'Profile 4' finished.
< undefined
> |
```

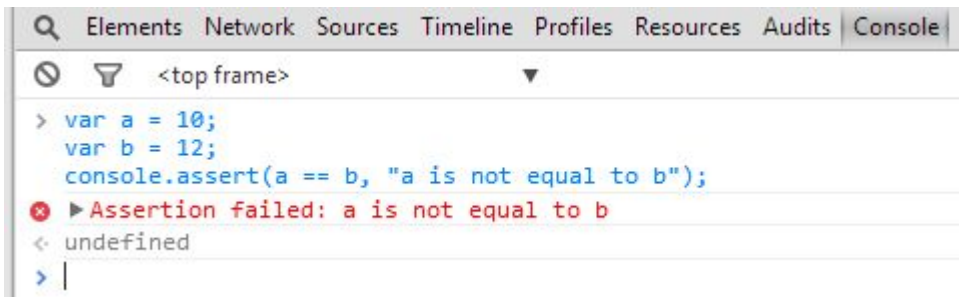


The screenshot shows the Chrome DevTools Profiler interface. The 'Profiles' tab is active, displaying a CPU profile for 'Profile 1'. The table below shows the breakdown of the profile's execution time.

Self	Total	Function
100.00%	100.00%	(anonymous function)
0%	100.00%	InjectedScript.evaluate
0%	100.00%	InjectedScript._evaluateAndWrap
0%	100.00%	InjectedScript._evaluateOn
0%	100.00%	evaluate

# Making Assertions

- Method `.assert()` allows to make assertions about conditions in our code.
- Syntax: `console.assert(condition, message);`
  - **condition** – boolean condition to test;
  - **message** – error message to output if condition is not met.



```
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
> var a = 10;
  var b = 12;
  console.assert(a == b, "a is not equal to b");
✖ ▶ Assertion failed: a is not equal to b
< undefined
> |
```

# Best Practices

- Assume your code will fail
- Log errors to the server
- You, not the browser, handle errors
- Identify where errors might occur
- Throw your own errors
- Distinguish fatal versus non-fatal errors
- Provide a debug mode



---

# Useful links

# Links

- JavaScript Errors on W3Schools.com:  
[http://www.w3schools.com/js/js\\_errors.asp](http://www.w3schools.com/js/js_errors.asp)
- Error object on MDN:  
[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Error](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error)
- Enterprise JavaScript Error Handling:  
<http://www.slideshare.net/nzakas/enterprise-javascript-error-handling-presentation>
- Debugging JavaScript in Google Chrome:  
<https://developer.chrome.com/devtools/docs/javascript-debugging#breakpoints-dynamic-javascript>

---

**Thank you!**