

[Lesson 5]

Roi Yehoshua 2018

[What we learnt last time?]

- What is loop?
- Types of loops in Javascript
- Breaking the loop
- Jumping to the next loop iteration

[Our targets for today]

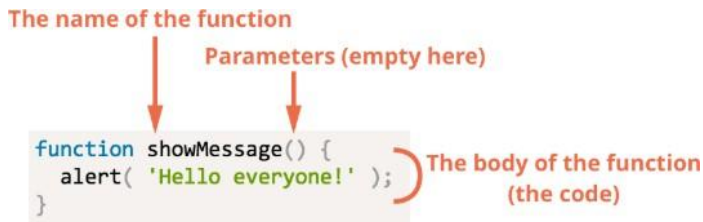
- Functions in Javascript
- Passing functions as arguments
- Scope of various types of variables
- Debugging Javascript code

[Functions]

- Quite often we need to perform a similar action in many places of the script
 - For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else
- Functions are the main “building blocks” of the program
- They allow the code to be called many times without repetition
- We’ve already seen examples of built-in functions, like `alert(message)` and `prompt(message, default)`, but we can create functions of our own as well

[Function Declaration]

→ To create a function we can use a *function declaration*:



The diagram shows a function declaration with three annotations: 'The name of the function' pointing to 'showMessage()', 'Parameters (empty here)' pointing to the empty parentheses '()', and 'The body of the function (the code)' pointing to the code block '{ alert('Hello everyone!'); }'.

```
function showMessage() {  
    alert( 'Hello everyone!' );  
}
```

→ Our new function can be called by its name: showMessage()

```
function showMessage() {  
    alert('Hello everyone!');  
}  
showMessage();  showMessage();
```

→ The call showMessage() executes the code of the function

→ This example clearly demonstrates one of the main purposes of functions: avoid code duplication

[Local Variables]

- A variable declared inside a function is only visible inside that function.
- For example:

```
function showMessage() {  
    let message = 'Hello, I'm JavaScript!'; // local variable  
    alert(message);  
}  
showMessage(); // Hello, I'm JavaScript!  
alert(message); // <-- Error! The variable is local to the function
```

[Global Variables]

- Variables declared outside of any function, are called *global*
- Global variables are visible from any function

```
let userName = 'John';  
function showMessage() {  
    let message = 'Hello, ' + userName; alert(message);  
}  
showMessage(); // Hello, John
```

- If a same-named variable is declared inside the function, it *shadows* the outer one:

```
let userName = 'John'; function showMessage() {  
    let userName = 'Bob'; // declare a local variable let message = 'Hello, ' +  
    userName; // Bob alert(message);  
}  
// the function will create and use its own userName  
showMessage();  
alert(userName); // John, unchanged, the function did not access the outer variable
```

[Global Variables]

- Usually, a function declares all variables specific to its task
- Global variables only store project-level data, so when it's important that these
- variables are accessible from anywhere
- Modern code has few or no globals
- Most variables reside in their functions

[Parameters]

- We can pass arbitrary data to functions using parameters (also called *function arguments*)
- In the example below, the function has two parameters: from and text

```
function showMessage(from, text) { // arguments: from, text
  alert(from + ': ' + text);
}

showMessage('Ann', 'Hello!'); // Ann: Hello!  showMessage('Ann', "What's
up?"); // Ann: What's up?
```

- When the function is called, the given values are copied to local variables from and text, i.e. the arguments are passed **by-value**

[Pass By Value]

→ If a function changes one of its parameters, the change is not seen outside, because a function always gets a copy of the value:

```
function showMessage(from, text) {  
  from = '*' + from + '*'; // make "from" look nicer  alert(from + ': ' +  
  text);  
}  
  
let from = 'Ann';  
showMessage(from, 'Hello'); // *Ann*: Hello  
  
// the value of "from" is the same, the function modified a local copy  
alert(from); // Ann
```

[Default Values]

- If a parameter is not provided, then its value becomes **undefined**
- For instance, the function `showMessage(from, text)` can be called with a single argument:
`showMessage('Ann');`
- That's not an error. Such a call would output "Ann: undefined"
- There's no text, so it's assumed that `text === undefined`
- If we want to use a "default" text in this case, then we can specify it after `=`:

```
function showMessage(from, text = 'no text given') {  
    alert(from + ": " + text);  
}  
  
showMessage('Ann'); // Ann: no text given
```

- Now if the text parameter is not passed, it will get the value "no text given"

[Default Parameters Old-Style]

- Old editions of JavaScript (before ES6) did not support default parameters
- There are alternative ways to support them, that you can find mostly in older scripts
- For instance, an explicit check for being undefined:

```
function showMessage(from, text) {  
  if (text === undefined) {  
    text = 'no text given';  
  }  
  
  alert(from + ": " + text);  
}
```

- Or the || operator:

```
function showMessage(from, text) {  
  // if text is falsy then text gets the "default"  
  value text = text || 'no text given';  
  ...  
}
```

[Returning a Value]

- A function can return a value back into the calling code as the result
- The simplest example would be a function that sums two values:

```
function sum(x, y) {  
    return x + y;  
}  
  
let result = sum(1, 2); alert(result); //3
```

- The directive **return** can be in any place of the function
- When the execution reaches it, the function stops, and the value is returned to the calling code

[Returning a Value]

- It is possible to use return without a value - that causes the function to exit immediately. For example:

```
function showMovie(age) {  
    if (age < 18)  
        return;  
  
    alert('Showing you the movie');  
}
```

- If a function does not return a value, it is the same as if it returns undefined:

```
function doNothing() { /* empty */ }  
  
alert(doNothing() === undefined); // true
```

[Naming Functions]

- A function name should clearly describe what the function does
- When we see a function call in the code, a good name instantly gives us an understanding what it does and returns
- A function is an action, thus it is a widespread practice to start a function with a verbal prefix which vaguely describes the action
- For instance, functions starting with...
 - "show..." – usually show something.
 - "get..." – return a value
 - "calc..." – calculate something
 - "create..." – create something
 - "check..." – check something and return a boolean

[One Function – One Action]

- Functions should be short and do exactly one thing
- Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two)
- A separate function is not only easier to test and debug – its very existence is a great comment!
- A few examples of breaking this rule:
 - `getAge` – would be bad if it shows an alert with the age (should only get)
 - `createForm` – would be bad if it modifies the document, adding a form to it (should only create it and return)
 - `checkPermission` – would be bad if displays the access granted/denied message (should only perform the check and return the result)

[Function Expressions]

→ The **function** keyword can be used to define a function inside an expression

```
let getRectArea = function (width, height) {  
    return width * height;  
}  
  
console.log(getRectArea(3, 4)); // 12
```

→ The function name can be omitted in function expression, in which case the function is **anonymous**

→ Function expressions in JavaScript are not hoisted, unlike function declarations, i.e., you can't use function expressions before you define them:

```
notHoisted(); // ReferenceError: notHoisted is not a function  
  
let notHoisted = function () { console.log('test');  
};
```

[Functions as Values]

- In JavaScript, a function is a value, so we can work with it like with other kinds of values
- For example, we can copy a function to another variable:

```
function sayHi() {      // (1) create alert('Hello');  
  }  
  
let func = sayHi;      // (2) copy  
  
func(); // Hello      // (3) run the copy (it works)!  
sayHi(); // Hello      // this still works too (why wouldn't it)
```

[Callback Functions]

- You can also pass functions as arguments to other functions
- For example, we will write a function `ask(question, yes, no)` with 3 parameters:
 - `question` – text of the question
 - `yes` - Function to run if the answer is “Yes”
 - `no` - Function to run if the answer is “No”
- The function asks the question and depending on the user's answer calls `yes()` or `no()`:

```
function ask(question, yes, no) { if (confirm(question)) yes() else
    no();
}
function showOk() { alert('You agreed.');
```

```
}
function showCancel() {
    alert('You canceled the execution.');
```

```
}
// usage: functions showOk, showCancel are passed as arguments to ask
ask('Do you agree?', showOk, showCancel);
```

[Callback Functions]

- The arguments of ask are called *callback functions* or just *callbacks*
- The idea is that we pass a function and expect it to be “called back” later if necessary
 - In our case, showOk becomes the callback for the “yes” answer, and showCancel for the “no”
- We can use Function Expressions to write the same function much shorter:

```
function ask(question, yes, no) { if (confirm(question))
    yes() else no();
}

ask(
    'Do you agree?',
    function() { alert('You agreed. '); },
    function() { alert('You canceled the execution. '); }
);
```

- Here, functions are declared right inside the ask(...) call. They have no name, and so are called *anonymous*. Such functions are not accessible outside of ask, but that’s just what we want here.

[Arrow Functions]

- There's one more very simple and concise syntax for creating functions, that's often better than Function Expressions
- It's called "arrow functions", because it looks like this:

```
let func = (arg1, arg2, ...argN) => expression
```

- This creates a function func that has arguments arg1...argN, evaluates the expression on the right side with their use and returns its result
- It's roughly the same as:

```
let func = function(arg1, arg2, ...argN) {  
    return expression;  
}
```

[Arrow Functions]

→ Example:

```
let sum = (a, b) => a + b;  
/* The arrow function is a shorter form of:  
  
let sum = function(a, b) { return a + b;  
};  
*/  
  
alert(sum(1, 2)); // 3
```

→ If we have only one argument, then parentheses can be omitted:

```
let double = n => n * 2;  
    // same as  
// let double = function(n) { return n * 2 }  
  
alert(double(3)); // 6
```

[Arrow Functions]

→ If there are no arguments, parentheses should be empty:

```
let sayHi = () => alert("Hello!");  
  
sayHi();
```

→ Arrow functions can also be used as callback functions:

```
function ask(question, yes, no) {  
  if (confirm(question)) yes()  
  else no();  
}  
  
ask(  
  'Do you agree?',  
  () => alert('You agreed.'),  
  () => alert('You canceled the execution.')  
);
```

[Multiline Arrow Functions]

- Sometimes arrow functions need to be a little bit more complex, like having multiple expressions or statements
- It is also possible, but we should enclose them in curly braces, and then use a normal return within them

```
let sum = (a, b) => { // the curly brace opens a multiline function
  let result = a + b;
  return result; // if we use curly braces, use return to get results
};

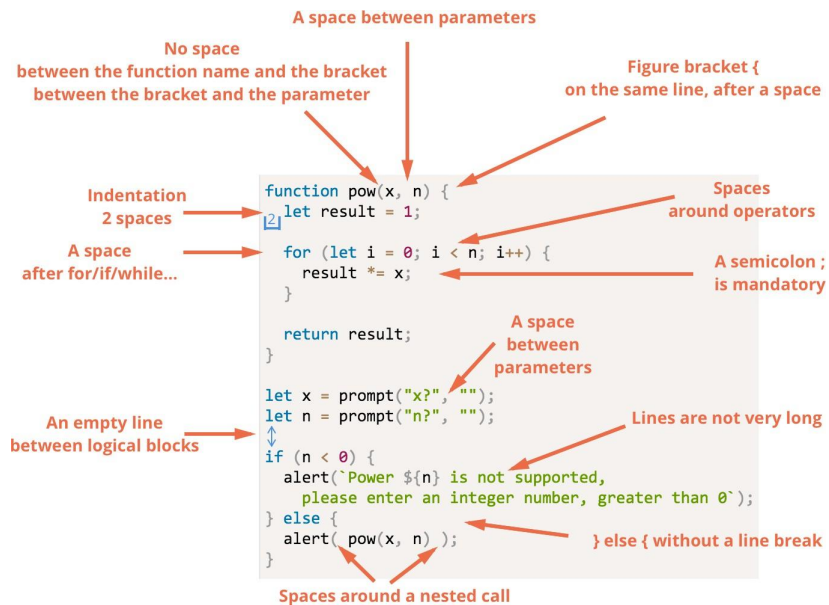
alert(sum(1, 2)); // 3
```


[Summary]

- Functions are values. They can be assigned, copied or declared in any place of the code.
- If the function is declared as a separate statement in the main code flow, that's called a "Function Declaration".
- If the function is created as a part of an expression, it's called a "Function Expression".
- Function Declarations are processed before the code block is executed. They are visible everywhere in the block.
- Function Expressions are created when the execution flow reaches them.
- We should use a Function Expression only when a Function Declaration is not fit for the task.
- Arrow functions are handy for one-liners. They come in two flavors:
 - Without curly braces: `(...args) => expression` – the right side is an expression: the function evaluates it and returns the result.
 - With curly braces: `(...args) => { body }` – brackets allow us to write multiple statements inside the function, but we need an explicit return to return something.

[Coding Style]

- Our code must be as clean and easy to read as possible
- You should follow the following coding style rules:



[Debugging in Chrome]

- All modern browsers support “debugging” – a special UI in developer tools that makes finding and fixing errors much easier
- We’ll be using Chrome here, because it’s probably the most feature-rich in this aspect
- Create the following example page and open it in Chrome:

```
function ask(question, yes, no) { if (confirm(question)) yes()
    else no();
}
function showOk() {
    alert('You agreed.');
```

```
}
function showCancel() {
    alert('You canceled the execution.');
```

```
}
// usage: functions showOk, showCancel are passed as arguments to
ask
ask('Do you agree?', showOk, showCancel);
```

[Debugging in Chrome]

- All modern browsers support “debugging” – a special UI in developer tools that makes finding and fixing errors much easier
- We’ll be using Chrome here, because it’s probably the most feature-rich in this aspect
- Create the following index.html page and hello.js script:

```
<!DOCTYPE html>
<html>
<head>
  <title></title>
</head>
<body>
  <script src="hello.js"></script>

  An example for debugging.

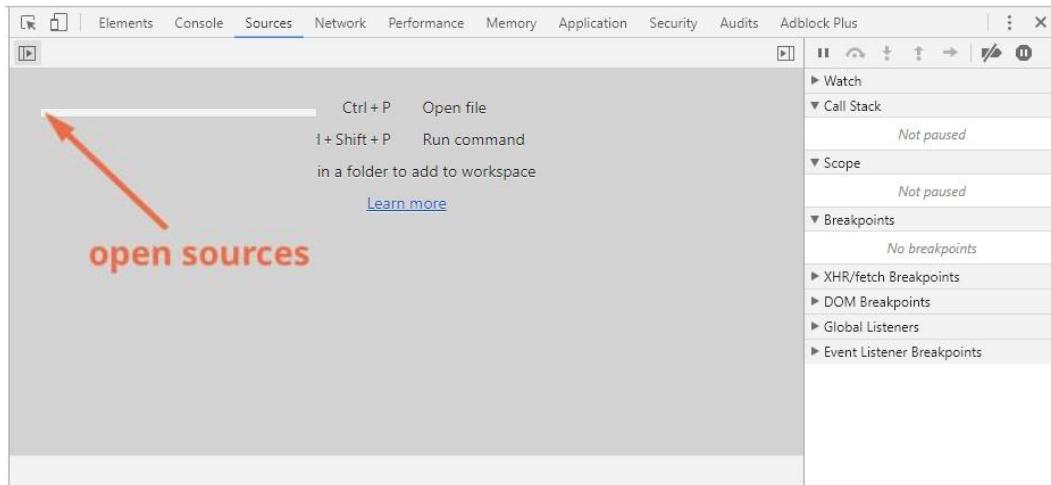
  <script>
    hello("John");
  </script>
</body>
</html>
```

```
// hello.js
function hello(name) {
  let phrase = `Hello, ${name}!`;
  say(phrase);
}

function say(phrase) { alert(`**
  ${phrase} **`);
}
```

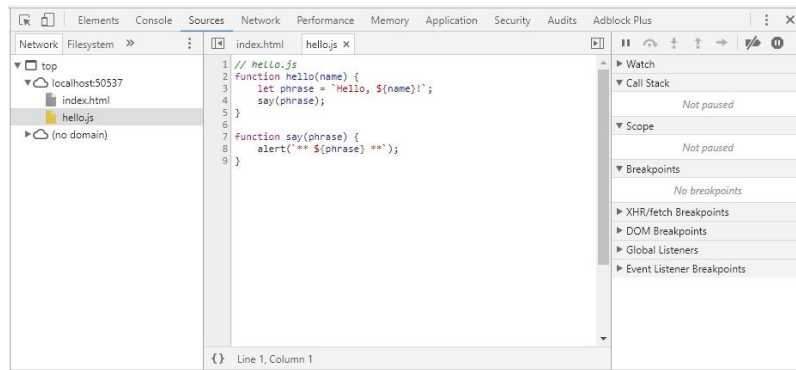
[Debugging in Chrome]

- Open the HTML page in Chrome
- Turn on developer tools with F12
- Select the sources pane
- Here's what you should see if you are doing it for the first time:



[Debugging in Chrome]

- The toggler button opens the tab with files
- Let's click it and select index.html and
- then hello.js in the tree view
- Here we can see three zones:
 - The **Resources zone** lists HTML, JavaScript, CSS and other files, including images that are attached to the page
 - The **Source zone** shows the source code
 - The **Information and control zone** is for debugging, we'll explore it soon
- Now you could click the same toggler again to hide the resources list and give the code some space



[Console]

- If we press Esc, then a console opens below
- We can type commands there and press Enter to execute
- After a statement is executed, its result is shown below.
- For example, here $1+2$ results in 3, and `hello("debugger")` returns nothing, so the result is undefined:

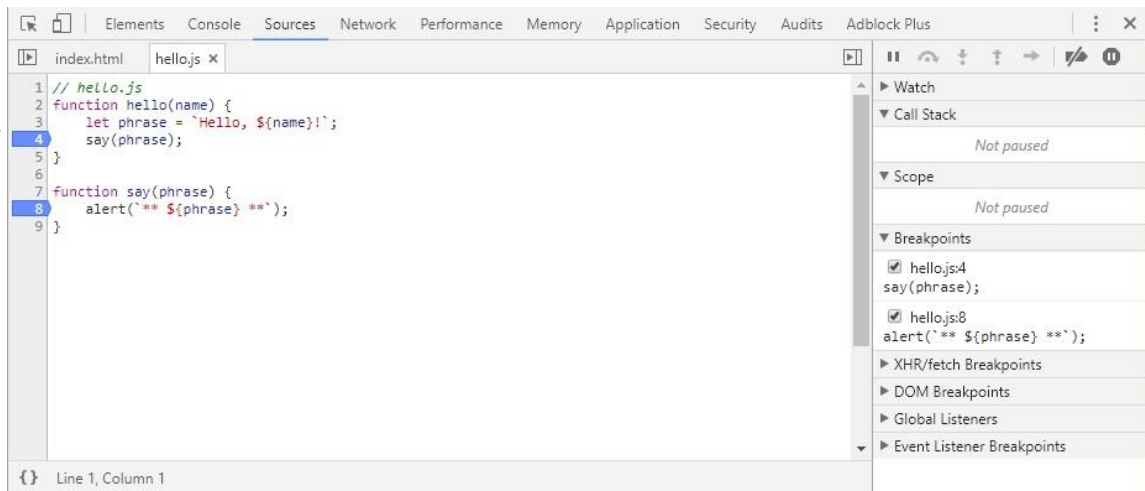


```
⋮ Console
▶ 🔇 top Filter Default levels ▾ Group similar
> 1 + 2
< 3
> hello("debugger")
< undefined
> |
```

[Breakpoints]

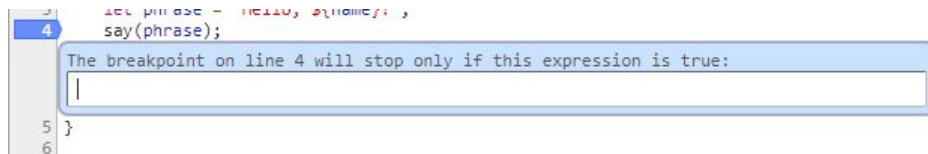
- Let's examine what's going on within the code of the example page
- In hello.js, click at line number 4. Yes, right on the 4 digit, not on the code.
- Congratulations! You've set a breakpoint. Please also click on the number for line 8.
- It should look like this (blue is where you should click):

breakpoints



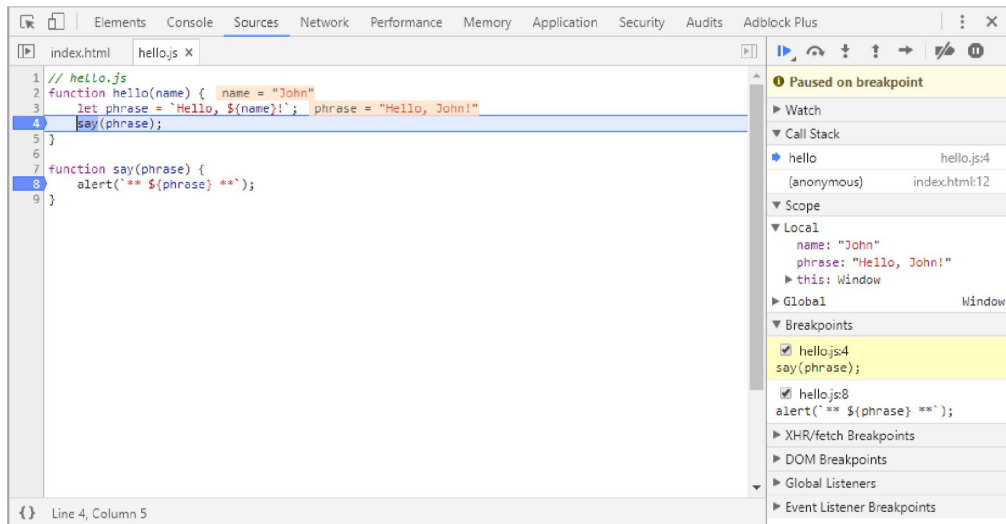
[Breakpoints]

- A **breakpoint** is a point of code where the debugger will automatically pause the JavaScript execution
- While the code is paused, we can examine current variables, execute commands in the console etc. In other words, we can debug it.
- We can always find a list of breakpoints in the right pane
- Right click on the line number allows to create a **conditional** breakpoint
 - It only triggers when the given expression is truthy
 - That's handy when we need to stop only for a certain variable value or for certain function parameters



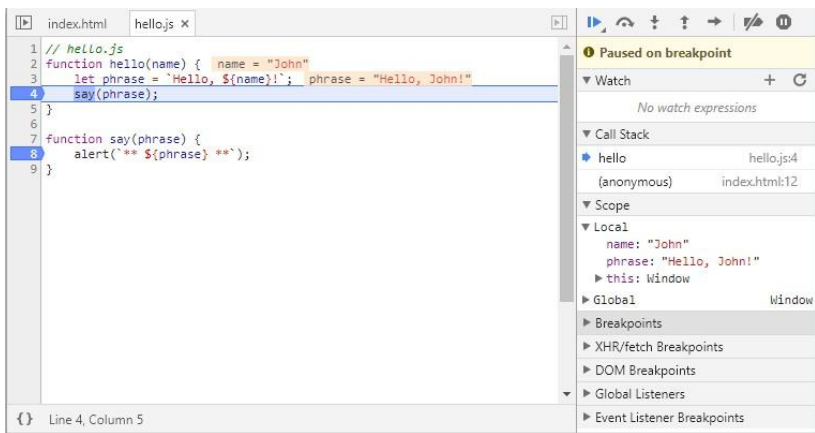
[Pause and Look Around]

- In our example, hello() is called during the page load, so the easiest way to activate the debugger is to reload the page.
- So let's press F5 (Windows, Linux) or Cmd+R (Mac)
- As the breakpoint is set, the execution pauses at the 4th line:



[Pause and Look Around]

- In our example, hello() is called during the page load, so the easiest way to activate the debugger is to reload the page.
- So let's press F5 (Windows, Linux) or Cmd+R (Mac)
- As the breakpoint is set, the execution pauses at the 4th line:



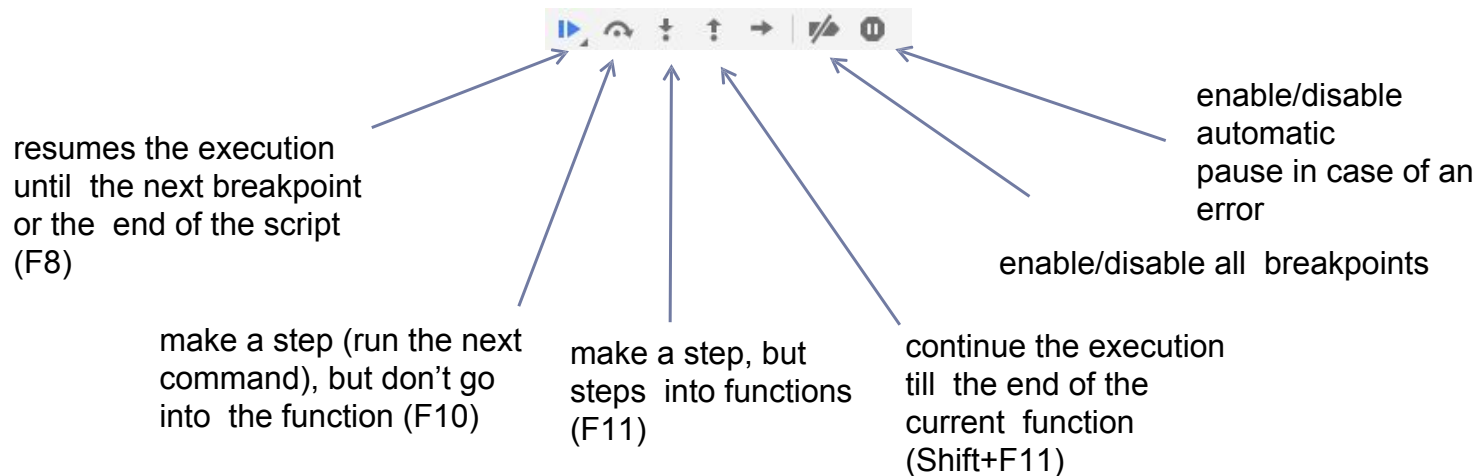
Watch – shows current values for any expressions that you enter

Call Stack – shows the nested calls chain. If you click on a stack item, the debugger jumps to the corresponding code

Scope – current variables. Local shows local function variables. Global has global variables (out of any functions).

[Tracing the Execution]

- Now it's time to *trace* the script
- There are buttons for it at the top of the right pane



[Control questions]

1. What is function?
2. What are the ways to declare a function in Javascript?
3. What is a variable scope?
4. What is a global scope?
5. How type of variable influences it's scope?
6. What is a callback?
7. What is arrow function?
8. How can you debug your code?

[Materials]

Core materials:

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Guide/Functions>

<https://learn.javascript.ru/function-basics>

<http://learn.javascript.ru/es-function>

[https://msdn.microsoft.com/ru-ru/library/bzt2dkta\(v=vs.94\).aspx](https://msdn.microsoft.com/ru-ru/library/bzt2dkta(v=vs.94).aspx)

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Statements/debugger>

<https://developer.mozilla.org/ru/docs/Web/API/Console>

Additional materials:

<https://learn.javascript.ru/global-object>

<https://learn.javascript.ru/recursion>

<https://learn.javascript.ru/debugging-chrome>

<https://developers.google.com/web/tools/chrome-devtools/javascript/>

Video materials:

<https://youtu.be/M3ACkFObsQw>

<https://youtu.be/R3kEcA8ZJNw>